July 1991

Report No. STAN-CS-91-1373

Thesis

PB96-149349

High-Performance Host Interfacing for Packet-Switched Networks

by

Hemant Ratubhai Kanakia

Department of Computer Science

Stanford University

Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1991	

4. TITLE AND SUBTITLE

High-Performance Host Interfacing for Packet-switched Networks

5. FUNDING NUMBERS

6. AUTHOR(S)

Hemant Ratubhai Kanakia

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Stanford University
Computer Science Department
Margaret Jacks Hall
Stanford, CA 94305-2140

8. PERFORMING ORGANIZATION REPORT NUMBER

STAN-CA-91-1373

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA
Arlington, VA 22209

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

High performance computer communication between users requires significant improvements over conventional host-to-network interfaces. Conventional host-to-network interfaces impose excessive processing, system bus and interrupt overhead on a host. We discuss in this report three closely related questions that should be addressed in designing a high performance host interface. The first key question is should one have to change transport protocols. The second key question is how to divide transport protocol processing between a front-end, that we shall call network adapter, and the host processor. The third key question is what are the important trade-offs in the design of a network adapter.

As a result of this investigation, we proposed a Network Adapter Board (NAB) architecture. A prototype for NAB was built and the performance for the prototype shows an order of magnitude higher throughout for large data transfer and almost a third lower latency for small amounts of data transfer.

Based on this work, we conclude that state machines of current transport protocols do not have to change for high performance. The only desired change is the streamlining of transport protocols to facilitate techniques such as pipelined processing, predictive header processing, and optimization of latency for small packets.

14. SUBJECT TERMS

15. NUMBER OF PAGES
53

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

HIGH-PERFORMANCE HOST INTERFACING FOR PACKET-SWITCHED NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Hemant Ratubhai Kanakia
November 1989

Preface

High performance computer communication between users requires significant improvements over conventional host-to-network interfaces. Current host-to-network interfaces impose excessive processing, system bus and interrupt overhead on a host. Current network adapters are either limited in function, wasting key host resources such as the system bus and the processors, or else intelligent but too slow because of complex transport protocols and inadequate internal memory architectures. Conventional transport protocols are difficult and costly to implement in hardware and too slow without it. With networks moving to the gigabit range, these problems will persist in spite of improvements in processor speeds and memory cycle time - unless significant design improvements are achieved.

We identified three closely related questions that should be addressed in designing a high performance host interface. The first key question is how to divide transport protocol processing between a front-end, which we shall call a network adapter, and the main processor. Adapters with sufficient intelligence to perform transport protocol processing are costly and the cost has to be balanced against the savings thus made in main processor cycles, host bus and memory bandwidth. The second key question is how much transport-level performance is available with an intelligent adapter. The third key question is how much, if any, change in conventional transport protocols is necessary for high performance.

We show that adapters that perform end-to-end checksums, data encryption, and packetization minimize data movement over the host bus and memory, which are critical host resources. Current intelligent adapters are slow and require rethinking of its architecture. We studied three different strategies for increasing the performance of intelligent adapters: pipelined processing, prediction-based header processing, and optimizing latency for packets with small amounts of data. Pipelined processing reduces per packet processing by folding the cost of checksumming and data encryption with that of moving data from the adapter to the network. Prediction-based header processing uses the hints generated by observing packets received in the near past to reduce packet processing time. Optimizing latency for packets with small amounts of data depends on recognizing and expediting these packets through the adapter. We integrated these three strategies in proposing a Network Adapter Board (NAB) architecture. The NAB architecture is designed to take advantage

of advances in processor technology. It implements few performance-critical functions in hardware using a general-purpose processor to handle the remaining functions. A prototype for NAB was built for the VMP system, a high-performance multiprocessor workstation developed at Stanford University. The performance for this prototype shows an order of magnitude higher throughput for large data transfer and almost a third lower latency for small amounts of data transfer.

Three conclusions, supported by our work, stand out. State machines of current transport protocols do not have to change for high performance. State machines of current transport protocol do not have to be implemented in hardware for high performance. The only required change is the streamlining of transport protocols to facilitate techniques such as pipelined processing, predictive header processing, and optimization of latency for small packets.

Acknowledgments

Throughout my apprenticeship at Stanford I have accumulated many personal and intellectual debts. By far the largest debt is to my advisor, David Cheriton. Through his persistent probing and incisive comments, he has had a large impact on both the content and presentation of this work. Quite apart from his direct influence on this work, I have many less tangible things to thank David for. By his own example and incisive comments, David taught me a great deal about organizing material for written and oral presentation, about constructive ways of dealing with criticisms, and about good scholarship. These are the lessons I value most.

Special thanks to the other members of Stanford Faculty involved with this work, John Hennessy, Geo Wiederhold, Richard Swanson, and Fouad Tobagi. John believed in me and encouraged me at a time when even I had doubts. Geo Wiederhold patiently listened and made valuable comments on my work. Although this work is outside his area of interest, Richard patiently read many drafts of this thesis without complaints. Fouad Tobagi, with whom I had the pleasure of working during my thesis work, has been in many ways instrumental for this thesis being what it is. From his example, I learnt invaluable lessons about academic life and the value of thoroughness in scholarship.

The students of Stanford Computer Science department and Computer Systems Laboratory have helped make my stay here a pleasant one. I have enjoyed discussions, movies, music, and playing volleyball with Pat Boyle, Steve Deering, Ross Finlayson, Cary Gray, Henk Goosen, Zygmond Haas, Kieran Harty, Shaibal Roy, Ashok Subramanium, Devika Subramanium, Arun Swami, Don Geddis, Michael 'R', Mehdi Nassehi, Jose Brazio, David Shur, Michael Fine, and Carey Williamson. My office has been a nice place to work thanks to Joe Pallas, Carey Williamson, and Karen Myers. One person in particular I owe a special debt to is Tsaki Birk, who has been a friend throughout my work and has been an excellent person to bounce off ideas.

Numerous others from the Stanford Community have contributed to my remaining sane through the dissertation process. Special thanks to Jitendra Malik, Isha Ray, Vikas Sonvalkar, Shankar Narayan, Bakul Shah, Sanjay Kasturia, Ashitava Ghoshal, Subroto Bose, Shashi Kanbur, Rajan Sekaran, Lucinda and Varadarajan Raghavan, all partners in late night sessions where one generally discussed meaning of life. Gopal Raghavan made the last few years in school endurable and even fun. Camille Marder helped in her own ways to see beyond the completion of my thesis. Kattie

Solomonson, having shared the similar experience in the dissertation process, became a friend with whom to compare notes, to moan and to wonder about the whole process.

Above all, I owe a special debt to Sonal Desai who endured all the ups and downs, listened patiently to problems and shared everything in life at Stanford. Rahul Kanakia, who arrived rather late on the scene, made life fun and had me look forward to the life with sweet anticipation. Finally, I owe a lot to my parents who pretty soon ceased to appreciate what I was doing at Stanford but nonetheless gave moral support whole-heartedly.

Contents

Preface iv

Acknowledgments vi

1 Introduction 1

1.1 Scope of Thesis . . . 3
1.2 Our Answers: In Brief . . . 4
1.3 Our Framework . . . 6
1.3.1 Processor and Memory Speed . . . 6
1.3.2 Simplification of Transport Protocols . . . 7
1.3.3 General-Purpose or Customized Hardware . . . 8
1.4 Outline of Thesis . . . 9

2 Measurements of Current Systems 10

2.1 Introduction . . . 10
2.2 Measurement Environment . . . 11
2.3 Observed User-level Performance . . . 13
2.4 Packet Processing at End-nodes . . . 15
2.5 Internal Audit of Packet Processing Cost . . . 18
2.6 Stand-alone Systems . . . 21
2.7 The Common Case . . . 26
2.8 Complexity of Transport Protocols . . . 27
2.9 Results . . . 28

3 Host Interface Architecture: Key Principles 30

3.1 Introduction . . . 30
3.2 Onboard Intelligence . . . 30
3.2.1 Minimization of Data Movement . . . 31
3.2.2 Reduction of Packet Processing Time . . . 36

3.2.3 Overload Protection 42
3.3 Alternatives 43
3.3.1 Larger Packet Size 43
3.3.2 Cut-through Adapters 44
3.3.3 End-to-end Checksums 46
3.4 Summary 47

4 NAB Architecture: Details **49**
4.1 Introduction 49
4.2 Host-Adapter Interface 49
4.2.1 Short Message Handling 51
4.2.2 Long Message Handling 52
4.2.3 Network Firewall 52
4.3 NAB: Hardware Architecture 53
4.3.1 The Buffer Memory 55
4.3.2 The Packet Processing Pipeline 60
4.4 NAB: Software 64
4.4.1 The On-board General-Purpose Processor 64
4.4.2 Host-Adapter Interface: Details 65
4.5 Summary 69

5 Performance Analysis **71**
5.1 Introduction 71
5.2 Delay for Short Packets 72
5.3 Throughput for Packet Groups 73
5.4 Effect of Improving Processor Technology 74
5.5 Cost/Performance Analysis 77
5.5.1 Packet Processing Pipeline 78
5.5.2 Memory Organization 79
5.5.3 Onboard Intelligence 81
5.6 A Summary of Results 83

6 Related Works **85**
6.1 Introduction 85
6.2 Better Software 86
6.3 Hardware Support for Protocols 87
6.4 Host-Network Device Interactions 88
6.5 Optimised Transport Protocols 89

6.6 Current Commercial Products 90

7 Conclusions and Future Work **92**

7.1 Conclusions 92

7.2 Future Work 96

Bibliography **99**

Chapter 1

Introduction

Performance of transport protocols on multi-megabit data communication networks tends to be limited by processing overhead at user nodes. For example, measurements of the V kernel [12] indicate that network transmission time on the Ethernet accounts for only about 20 percent of the elapsed time for transport-level communication operations, even with its highly optimized protocol. Similar performance figures have been reported elsewhere [41, 49]. Although processor and memory cycle times keep improving, with communication networks moving to gigabit range, we expect the processing to persist as a bottleneck unless significant improvements in network adapter and transport protocol designs are achieved. This research focuses on understanding and eliminating this processing bottleneck.

In this thesis, the term "host interface" denotes the hardware and software used at the host and at the network device to communicate over a packet-switched network. The term "network device" or "adapter" refers to the subsystem dedicated to communication-related tasks.

We identify three specific problems with current designs of host interfaces. The host interface imposes excessive overhead in the form of processor cycles, bus bandwidth and host interrupts. The overhead arises from checksumming, packetization, and encryption of data performed at transport level. The memory-intensive processing required by these functions reduces the average instruction execution rate. This is a more important effect for high-performance processors, such as MIPS [29], in which memory reference operations are proportionally much slower than register-only operations. This processing causes the data to move at least twice over the system bus – once from global memory to the processor (or its cache) and once when the packet is copied to the network adapter. This increased traffic wastes the system bus bandwidth, which is a critical resource in both multiprocessor and uniprocessor machines. In current host interfaces, the host is interrupted for each packet received or transmitted. These interrupts force frequent context switching with the attendant overheads.[1] This imposes a high penalty in the multiprocessor system with processor caches, where it may be necessary to fault code and data into the cache before responding to the interrupt. In addition, the context switch may also incur contention overhead when data associated with the network module is resident in another cache. The problem is further aggravated by the prospect of networks moving to the 100 megabit up to the gigabit range using fiber optics [24, 28, 44]. For instance, in a file server attached to 1 Gigabit network with the interface interrupting on every 2 KBytes[2] packet, the network interrupts every 200 microseconds under load.

An intelligent adapter offers a solution to these problems. It reduces data movement over the host bus, saves main processor cycles, and minimizes host

[1] A network device driver could try to avoid full context switch for each packet interrupt. Recently, researchers from DEC's Systems Research Center at Palo Alto [60] did built such a driver. In this device driver, a packet is processed by a low-level interrupt handler and queued for a user but actual notification and waking up of a user process is delayed till later. It was reported that although the throughput improved with such a driver, when the low-level interrupt was extended to do network or transport level header processing, the driver became complex and slow.

[2] Since most current applications use packets which are equal to or smaller than 2 KBytes, in reality we will continue to have small size packets even on high-speed networks. Moreover, increasing the size of a packet has many non-obvious disadvantages, such as increasing latency of users with small packets, increasing buffers required at switches, gateways, and host interfaces, and increasing jitter and loss rate in the network. Thus, we should not engineer networks or host interfaces to work optimally with large size packets.

interrupts. We use the term "intelligent" adapter to indicate that it implements transport protocol functions.

The main problem is that current intelligent adapters offer low throughput and high response times. The problem motivated us to investigate techniques for increasing performance of intelligent adapters. Preliminary measurements indicated that the primary reason is an inadequate internal memory architecture. Currently, the data transfers into and out of the buffer memory reduces the number of memory cycles available for packet processing. As we progress to a gigabit range network, this problem will become even more acute.

Conventional transport protocols are too complex or awkward for hardware implementation (and too slow without it). But since the packet processing cost is concentrated in few routines, hardware implementation of these routines would substantially increase performance. For a large packet, the processing cost incurred in checksumming and encryption dominates the packet processing since the cost increases in proportion to the size of a packet. Although hardware implementation for these functions would increase performance, conventional transport protocols are designed with little or no thought given to facilitating hardware support or implementations For increasing performance, there is little reason to redesign state machines for conventional protocols. The state machine specifies procedures for processing headers, which typically is a small part of the overall cost of packet processing. Furthermore, with the predictive processing technique described in Section 3.2.2, the cost of header processing remains mostly independent of the complexity of the state machine.

An additional factor that motivates redesign of network adapter architectures is the problem of a host being bombarded by packets from one or more other hosts. The packet arrival rate, especially in a high-speed network, can exceed the rate at which a host can process and discard these packets, effectively incapacitating the host for useful computation. Excessive packet traffic can arise either from failures or malicious host behavior. A well-designed network adapter acts as a "firewall" between the network and the host.

1.1 Scope of Thesis

The primary goal of this thesis is to achieve a deeper understanding of performance issues which would guide future designs of protocol and host system architecture. The scope is best defined by stating the three closely related questions addressed in this thesis research.

Question 1:

How much of transport-level protocol processing should be done at an adapter?

Question 2:

How to increase performance of adapters performing transport-level functions?

Question 3:

Are changes in conventional transport protocols necessary for increasing performance?

To address these question concretely, we used a paradigm that is typical for systems research. First, we measured the performance of existing systems in order to ferret out "real" bottlenecks. Based on the observations thus made, we proposed a solution to remove these bottlenecks. We designed and partially implemented a prototype host interface and analyzed its performance. The experience thus gained in turn strengthened the answers we provided to these questions.

1.2 Our Answers: In Brief

We propose that an adapter perform end-to-end checksums, encryption/decryption and packet processing of *intermediate*[3] packets. This choice will minimize data movement on the host bus, and save main processor cycles. An additional benefit of having onboard intelligence is the protection it offers to a host against network malfunctions and hostile remote users.

Three techniques we studied in order to increase performance of intelligent adapters are the pipelined processing, prediction of headers, and minimization of latency for small packets. The *pipelined processing* of a single packet involves performing checksumming and encryption functions while the packet is transmitted or received. The pipelined processing of multiple packets involves header processing of the ith packet concurrently with the transfer of data for the $i-1$th packet from/to host and the transfer of

[3] An intermediate packet is one that does not implicitly or explicitly establish or close a virtual connection.

the $i+1$st packet to/from network. We distinguish the later as the "concurrent processing" or the "overlapped processing" of packets.

Predictions of headers of the expected packets reduces packet processing times if the packets are actually received. We extended the header prediction technique, first proposed by V. Jacobson [37], for the request-response model observed in a typical distributed systems environment. The success rate for predictions depends on how many predicted headers are stored and on the locality in network traffic [62]. In current host interfaces, header predictions do not significantly increase performance [37]. But when the pipelined processing is used, the header predictions become more effective due to the overlap between header processing and data-related operations.

In a typical distributed systems environment, about 40% packets carry small amounts of data and are due to occasional packet exchanges between users. Minimizing latency for such packets is important for performance of distributed systems. Ideally, the host-to-adapter interface and packet processing software of the adapter should recognize and expedite such packets as a special case.

Based on these principles, we designed an intelligent adapter for the VMP multiprocessor [10], focusing on the architectural issues in the host-to-adapter interface, the internal architecture of the adapter board, and the transport protocol. The interface between adapter and a host is designed for minimal latency, minimal interrupt processing overhead, and minimal data transfer on the system bus.

The proposed architecture, called Network Adapter Board (NAB), encompasses three distinct areas: adapter hardware, adapter software, and host-to-adapter interactions. The adapter's system architecture has a novel internal memory and processing architecture that implements some of the key performance-critical transport layer functions in hardware. Its software utilizes locality observed in network traffic to reduce processing of received packets. In the NAB, we used a different host-device interaction model to reduce the data movements, and the waste of host resources.

The architecture is coupled to a new transport protocol called *Versatile Message Transaction Protocol (VMTP)* [6, 9]. VMTP is a request-response transport protocol specifically designed to facilitate implementation by a high-performance network adapter. VMTP assumes an underlying network (or internetwork) service providing a datagram packet service. VMTP provides a support for multicast datagram service, real-time data transfer, and secure data transfer. The interested reader is referred to an Internet RFC paper [9] for further details.

A prototype of the architecture was designed and partially implemented. The prototype is a single board connecting VMP multiprocessor nodes via a 100 MBits/s network link. It is estimated to sustain the effective throughput of 45 MBits/s for reliable transfer of 16 KBytes of user data, including the kernel overhead. The same operations on current host interfaces are done with the effective throughput of 4.2 MBits/s. Based on our finding that the processor used in the prototype was the bottleneck, we used a more powerful processor to study how much its performance improves. Rather than build another prototype, we used a simulation of AMD's Streamlined Instruction Processor (AMD29000), a RISC processor with a nominal rating of 17 MIPS, to model the NAB hardware. Running the NAB Software on this simulator, we observed effective throughput for 16-KByte read was about 400 MBits/s. The experiment assumed that the network link was a gigabit link. In summary, we found the architecture provides impressive gains compared to current software implementations.

1.3 Our Framework

We discuss in this section the trends in technology development, transport protocol design, and the cost of hardware development defining the framework for the thesis.

1.3.1 Processor and Memory Speed

The instruction execution time of general-purpose processors has continued to decrease more rapidly than cycle times of main memory. Instruction cycle time of early microprocessors such as 8008 was roughly equal to cycle time of memories available at that time. Currently, with the development of RISC technology, which uses a reduced instruction set in order to reduce instruction execution time, the cycle times of 100 ns to 20 ns are achievable. During the same period, memory cycle times have decreased rather slowly to go from 1-2 μs to 120 ns (for a dynamic RAM of moderate size). The gap in demand and supply of memory bandwidth is particularly se-

vere if we consider the memory bandwidth used up by high-speed I/O traffic. The future holds no promise for reducing this gap. Changes in processor designs are aimed towards lower and lower instruction cycle times [22, 31], whereas DRAM technology is aimed towards increasing the size and not the cycle time. Moreover, although the basic memory cell is becoming smaller and faster due to its redesign, the electrical limitations of a large cell arrays constitutes a major hurdle to further speed improvement. Drive and sense circuits for a large number of memory cells limit the rise and fall times of these pulses and their width. We expect the memory bandwidth limitation to become an increasingly significant factor in computer system designs, including the design of network devices.

Limited memory bandwidth has forced computer system designers to consider a hierarchical memory architecture where a cache (which is faster and smaller than main memory) serves as the intermediate store for data and instructions. What makes caching work well in practice is the observed locality in memory references of a typical program. It is used to reduce the effective instruction (and data) fetch time. Similarly, we can expect the limited memory bandwidth to change the design of a high speed network adapter.

1.3.2 Simplification of Transport Protocols

Light-weight transport protocols [14, 27] have been typically proposed to accommodate their implementation in hardware. Advocates of such protocols argue that simplification would lead to better performance at the transport protocol layer. The above conclusion is not supported by current evidence. On the contrary, various studies which audit packet processing time suggest that simplification is unlikely to affect performance. In [37], Van Jacobson reports on packet processing times observed for TCP/IP software on Unix running on a SUN workstation. Of the total 1500 microseconds spent processing a typical packet, he found that only about 16% of the time was taken by protocol-specific tasks. The rest was spent in the operating system and the movement of data. Another study of TCP software [17] also supports this observation. [41] shows that even with highly optimized data movement in a stand-alone environment, no more than 15% of the time is spent in performing IEEE 802.2 protocol-specific tasks. Hence simplifying a protocol addresses only a small part of the performance problem.

Recent measurement studies also point out another fact that undermines the simplification philosophy. As reported in [17, 36], the common case of processing a packet delivered without error requires few instructions. The complexity of protocol specifications is introduced due to exceptions. Processing cost of these exceptions, which occur infrequently, has little impact on the average performance of the transport protocol. Nor can we avoid this processing without dropping or altering the quality of transport-level service. Changes in algorithm of error and flow control schemes have no impact on the processing cycles required. For example, go-back-by-N retransmission scheme and selective retransmission both require an almost equal number of instructions per packet. The same is true for processing costs of sliding window versus rate-based flow control schemes.

We used a conventional transport protocol with minimum changes necessary for high performance.

1.3.3 General-Purpose or Customized Hardware

Recent efforts in high performance protocol implementation fall into two extremes: some use standard protocols and adapters but improve software implementations (we will call this a status-quo approach); the others simplify a protocol to have it implemented entirely in VLSI (we call this a silicon approach).

The status-quo approach depends on improvements in microprocessor technology and efficient software to get better performance. Its advocates cite adaptability to ever changing standards as the major strength of the approach. New standards are emerging at a rapid rate and even well-conceived standards seem to require a number of subtle modifications before they become stable. There is one other comparative advantage of this approach. General-purpose microprocessors receive more attention and more investment of resources to continue their rapid improvements. This could eventually overtake any short-term benefits provided by using a custom VLSI.

Advocates of the silicon approach like to point out that the improvements we can expect from the status quo approach are limited. The best long-term solution, they argue, is to adapt protocols and hardware for efficient processing of packets. They con-

clude that, since the functions needed are not significantly different than other types of I/O traffic, we should be able to design hardware similar to an I/O processor and a disk controller.

We propose a middle path between these two extremes. We believe that a better solution is to design hardware support for the common and the critical functions, leaving software to handle exceptions. This approach is based on our observations that the complexity of protocol exists mainly in the exception handling procedures, and that the changes in protocol specification occur frequently in these procedures. Typically, the frequently executed portion of the protocol specification remains stable over the years. Such key functions can and should be supported in hardware. We perceive that the real issue to be finding an appropriate boundary between hardware and software. Our approach integrates advantages of both extreme approaches. It has flexibility because it executes complex (and infrequent) tasks in software. It provides hardware support for fast execution of common functions.

1.4 Outline of Thesis

The first task carried out for this research was to measure existing implementations of protocols. The results of this study and its interpretation are recorded in chapter 2. The study motivated a design of the host interface for the VMP multiprocessor [10], focusing on the architectural issues in the host to adapter interface, the adapter board, and the transport protocol. In chapter 3, we address the question of why we need intelligent adapters and how they can be designed for high performance. In chapter 4, we describe the details of the internal architecture of the network adapter board and its interface to host. In chapter 5, we discuss results of performance measurements on the prototype. In this chapter, we also discuss cost/performance benefits of the pipelined processing and onboard intelligence. In chapter 6 we compare our design to other related works reported in literature. In chapter 7, we conclude with a summary and a discussion about future work.

Chapter 2

Measurements of Current Systems

2.1 Introduction

In this chapter, we discuss results of measurements of various transport protocol implementations. The primary goal is to understand how packet processing time is divided among functions performed. The understanding will point to key issues in designing network interfaces for high-speed networks. We begin here by outlining methods used to collect data in this study. Here we also discuss important features of VMTP transport protocol, used in distributed systems research at Stanford University. We observed performance of VMTP at user-level for different workstations and operating systems. We also compared performance of different transport protocols to avoid drawing conclusions from a study of a specific transport protocol. The observations confirm the well-known fact that user-level performance is lower than the transmission capacity of the network link. We focus next on measuring the total CPU time spent processing packets and compare the CPU time with the transmission time and the total response time for small and large amounts of data exchange in VMTP. Results show that CPU time is the major component of the total response time. Next we analyze various components of the CPU time spent processing a single packet. This is the crux of our measurement work and it provides valuable insights on what performance bottlenecks exists. These insights and its impact on the design are discussed in the final section of the chapter. These insights could be affected by the idiosyncrasies of an operating system. Recognizing this we also cite supporting evidence from results our study [41] of a dedicated and stand-alone implementation. One useful technique for increasing performance is to concentrate resources on executing the common case. Our measurements show what is the common case in packet processing and also how frequent it is. We also compared CPU times of various alternative error control schemes to support one of the conclusions drawn from this study, namely, that the complexity of a transport protocol is not a significant factor in performance. We conclude the chapter with a summary of conclusions based on observations of these measurements.

2.2 Measurement Environment

We have a distributed environment characterized by over 50 diskless workstations, four file server machines and a couple of print servers. The network is a single-segment 10-MB Ethernet connected to other Ethernets on the campus with bridges and gateways to Arpanet. Workstations are VAX stations, Sun 3's and 4's, few experimental multiprocessor workstations from Digital Equipment Corporation and a dual-cpu VAX 8350. All of the workstations and file servers run the V distributed operating designed at Stanford University. Inter-process communications in V is based on a message-based paradigm. The operating system provides a location-transparent mechanism to communicate with processes on multiple machines or a local machine. It is organized as a simple kernel that provides basic message-passing primitives. All of the rest of traditional operating system functions such as management of memory, I/O devices, and files is provided with various servers. Primary use of our environment is distributed systems research. It is used for compiling and editing programs as well as for daily chores such as reading mail and news, writing and formatting papers, and playing games such as multi-user maze-wars or checkers.

Message communications over the network uses Versatile Message Transport Protocol (VMTP) supporting basic RPC call mechanism and reliable transfers of bulk data. The main advantage that VMTP has with respect to transport protocols such as TCP and TP is its excellent fit with the requirements of distributed applications. The basic model of VMTP, referred to as request-response model, takes advantage of the observation that in most applications one expects a response (from the the application-level) to a request being made. VMTP uses this basic message exchange as a basis for providing reliable service. Other services such as uni-directional data transfer (datagrams), data transfer to a group of destinations (multicasts), and bulk data transfer (virtual circuit) are also implemented with this model. A message in TCP requires a separate acknowledgment from the transport layer and a separate response from the application. In contrast, VMTP reduces packets required in reliable data transfer by using a response to the requested service itself as an acknowledgment of transport-level packet. Request and response messages may be large requiring multiple packets for full transmission. Sequencing information necessary to reassemble a message from packets is provided. Packets formed out of a single message are divided into groups of 16 packets. Each group is sent out with an inter-packet gap specified by the sender. The cost of resource allocations for long-term communications is reduced by caching state information (required to prevent duplicates and to retransmit) across multiple transactions. In other words, all other types of communication patterns are built on top of this basic transaction model.

VMTP procedures minimize computations and state information required at the server end. This helps to keep VMTP implementation at server end simple and efficient, which allows multiple clients to use a server without it becoming the bottleneck. The protocol provides multicast request service to higher layers. VMTP also provides domain-based addressing and facilities to handle secure transmissions. VMTP uses selective retransmission and rate-based flow control to handle packet losses and congestion problems. The protocol is completely specified in [9] and its advantages are discussed in detail in [7, 6, 11].

Two important cases of end-user performance are response time for small amounts of data and throughput for bulk data transfers. The response time is the elapsed time between sending a request message and receiving the response message, including the overhead in the operating system kernel. The throughput is the effective data rate obtained by dividing user-level data transferred by the response time observed by a client. Performance of many distributed applications depends on the performance of both cases. For instance, a high-performance distributed file server would require low response time for messages exchanged during opening, closing and locking of a file as well as high throughput for read and write operations.

Bimodal distribution of packet sizes has been well-established [11]. In our network traffic, we found two distinct peaks of packet sizes, 64 and 1024 bytes. A large number of transactions require a very small amounts of data to be sent or returned, e.g., commands typed to the V executive (typically 1-30 bytes), host or user names (5-20 bytes), and host status information (usually 72 bytes). In one study of network measurements for VMTP traffic [11], it was found that the ratio of small to large packets is about 3:7. This observations have prompted VMTP (and the host-NAB interface) to distinguish small packets from large ones. Small packets are fixed in size and carry up to 64 bytes of user data. The distinction helps storage management and packet processing in software implementations. The distinction is also used in our proposed architecture to optimize response time.

2.3 User-level Performance

In Figure 2.1, we show the performance observed with current software implementations of VMTP. The data is obtained by making the same read (or write) call 1000 times and averaging over 10 trials. During these measurements, the background network traffic was measured to be about 25 packets/second. The throughput increases as we increase the data segment size, but it levels off after data segment size of 16 Kbytes, after which it improved no more than 5%.

In Figure 2.2, we compare response time for single packet exchange and throughput for 16 Kbytes through the 10-MB Ethernet with other three other cases: in-memory copy of data, local message passing, and transfer time over raw Ethernet. Memory transfers and local transfers between processes were measured on a Sun 3/50 workstation running V. The comparison shows the user gets less than one third, one tenth and one eleventh of throughput available,

Machine-Pair	Response Time 0 Bytes ms	Throughput 16 Kbytes Kbits/s
VAX-VAX	6.585	2132.7
VAX-SUN2	6.238	2291.0
VAX-SUN3	3.988	1586.6
SUN2-SUN2	6.472	1027.4
SUN2-VAX	6.275	2106.6
SUN2-SUN3	4.849	1310.0
SUN3-SUN3	2.933	3574.0
SUN3-VAX	4.165	2301.1
SUN3-SUN2	4.390	2457.7

Figure 2.1: User-level Performance of VMTP on Ethernet

	Response Time 0 Bytes ms	Throughput 16 Kbytes Mbits/s
Data copy	0.038	33.59
Local send	0.680	29.82
Remote send	2.933	3.57
Network delay	0.062	10

Figure 2.2: Overhead of VMTP as compared to in-memory data transfer and network transmission time

respectively, for network transfer, local message passing, and in-memory data copying. The response time of transport-level transfer is about 47, 4, and 77 times higher than, respectively, for network transfer, local message passing, and in-memory data copying. These measurements were made when background traffic was less than 25 packets/s. During the measurements itself, the network load remained less than 12% of theoretical maximum. This data clearly shows that packet processing is the most important component of cost in network communications.

Optimized design of an operating system or transport protocols may yield a factor of 2 improvement in performance. In Figure 2.3, we show the performance measured for VMTP under V and VMTP under Unix (4.3 BSD version) operating systems. Both were performed on the same Sun 3/75 workstation machine. In V, we get lower response time and high throughput as compared to Unix. In Figure 2.4, we show the measured performance of VMTP, TCP, SunRPC, and

Operating System	Response Time 0 Bytes ms	Throughput 16 Kbytes Mbits/s
V (Stanford)	2.93	3.574
Unix 4.3BSD	5.70	2.512

Figure 2.3: Performance of VMTP under V and Unix operating systems

Protocols	Response Time 0 Bytes ms	Maximum Throughput for (.) in Kbytes Mbits/s
VMTP	5.2	2.592 (16)
TCP	6.9	2.256 (2)
SunRPC	9.2	1.880 (8)
UDP	6.4	2.608 (8)

Figure 2.4: Performance of different transport protocols under Unix

UDP (a "simple" transport protocol) under Unix on a Sun 3/75 workstation machine. In brackets in second column we have indicated the user data segment size for which maximum throughput was observed. In all of these transfers, the network used was 10-MB Ethernet, which has the maximum packet size of 1536 bytes of data. We observe that VMTP offers lowest response time and highest throughput. The difference separating the performances is less than a factor of 2 in all cases.

2.4 Packet Processing at End-nodes

The host processor remains quite busy while sending and receiving messages. In Figures 2.5 and 2.6 , we show, respectively, the host processing power expended in communications-related tasks for small and large amounts of data exchange. The host machines were two Sun 3/50 workstations running V. For bulk data, each experiment consisted of 1000 calls for reading of 16 Kbytes. The measured total times were divided by 1000 to obtain per message time. The figures show the result of averaging over 10 such experiments. Similar set of experiments were conducted to obtain data for exchange of one packet

16 Kbyte Data Transfer	Client Time (ms)	Server Time (ms)
CPU	23.671	22.783
Elapsed	36.738	36.738
Ethernet	14.168	14.168

Figure 2.5: Measured CPU and transmission times for 16 Kbyte write operation in VMTP under V

0 Kbyte Data Transfer	Client Time (ms)	Server (ms)
CPU	1.804	1.775
Elapsed	2.933	2.933
Ethernet	0.125	0.125

Figure 2.6: Measured CPU and transmission times for write operation with no data

request-response transaction requesting null service. The transmission time on Ethernet is measured with an oscilloscope connected at the network-end of the client. For bulk data transfer, about 64% of the elapsed time consists of time used for packet processing. A slightly lower percentage (61%) is seen for the case of simple transactions. We also observe that packet processing time in both cases is almost equal at client and server. Only about 38% of the elapsed time is Ethernet transmission, confirming again that packet processing is the bottleneck. Note that since packet transmission overlaps with packet processing, addition of first two rows does not give the third one, the total elapsed time.

In Figures 2.7 and 2.8, we have compared the CPU cost to the moving of equivalent amounts of data in memory by CPU, and making the same call to a local process. Data has to be moved at least once during message passing to local or to a remote process. Thus, the cost of moving data is included in the later two costs. The comparison of costs suggests that most of the packet processing cost is the processing by the host. The relative cost of various functions performed for network transfer is analyzed in the next section.

CPU Time for Transfer of 16 Kbytes	Client ms	Server ms
In-memory data copy	3.902	3.902
Local VMTP send	4.395	4.314
Remote VMTP send	23.671	22.783

Figure 2.7: A comparison of processing costs for local and remote send with in-memory copy: 16 Kbyte data

CPU Time Measured for 0 Kbyte	Client ms	Server ms
In-memory data copy	0.038	0.038
Local VMTP send	0.680	0.680
Remote VMTP send	1.804	1.775

Figure 2.8: A comparison of processing costs for local and remote send with in-memory copy: No data

2.5 Internal Audit of Packet Processing Cost

Protocol-related processing is a minor part of the total processing time of a packet. This fact in revealed in any internal audit of packet processing. Figure 2.9 shows relative cost of various functions executed for VMTP implementation on a Sun 3/75 workstation. The costs were calculated by counting instructions executed in processing an error-free transmission of a packet. The Ethernet time is calculated using an oscilloscope attached to the network end of a client workstation. Only 7% is unaccounted when the total estimated time is compared with the measured per packet processing time.

In this figure, we observe that the time for processing VMTP and IP headers is less than 12% of the total time. The most significant part of the cost is the operating system overhead in processing a packet. This overhead consists of buffer management, interrupt processing, waking up the destination process, and device driver. The rest is divided equally between checksumming, moving of data from user to kernel space, and DMA time by the network device.

Relative costs remain constant for different operating systems. Figures 2.10-2.12 show results of the

Components	Processing Time
Data Movement	40 %
Kernel	25 %
Ethernet Driver	15 %
VMTP	8 %
Checksum	10 %

Figure 2.9: Relative costs in processing a VMTP packet under V running on a Sun 3/50 workstation while performing 16 Kbyte Writes

study of packet processing costs for VMTP on Unix. In Figure 2.10, we show relative costs of various functions executed in performing RPC call for a client machine, Sun 3/50 (column 1) and sun 3/160 (column 2). The measurements are collected using the profile measurement tool available on Unix. The experiment sent 100,000 RPC calls over a relatively unloaded[1] network. In Figure 2.11, we show relative costs under a similar experimental set-up for a Sun 3/75 and Sun 3/150 servers. In Figure 2.12, we show processing times measured for 100,000 transactions of 16 Kbytes read call using VMTP on Unix. We observe in these three figures that the major part of time spent in protocol-related functions and relatively large time spent is again due to the operating system overhead. Another study, performed by Van Jacobson of Lawrence Berkeley Laboratory [36], also confirms these conclusions for TCP/IP processing on a Sun 3 workstation running Unix (4.3 BSD Version). The relative costs reported in that study are shown in Figure 2.13.

2.6 Stand-alone Systems

Packet processing costs are divided into somewhat different components in a stand-alone system such as a network adapter. Such systems have a simple operating system which reduces its effect on packet processing rates. To study this, we measured the performance of an implementation of IEEE Std. 802.2 Logical Link Control protocol [56], which was chosen as a specific example to study because of our interest in a local network. It provides two types of services: reliable virtual connections and datagrams.

[1] Background packet traffic was not reported in the study [48], from where these measurements are quoted.

Read Operation (0 Bytes)	Sun3/50	Sun3/160
System call	5%	4.6% (16)
Socket interface	6.4%	6.0%
VMTP	26.1%	22.9%
Sleep, Wakeup	7.6%	5.1%
IP time	9.8%	9.4%
Software int	1.9%	1.8%
Others	3.3%	2.1%
Unaccounted	3.0%	0.5%
Total	99.1%	98.1%

Figure 2.10: Components of client-side processing for small data exchange in a VMTP implementation on Unix

Read Operation (0 Bytes)	Sun3/50	Sun3/160
System call	9.0%	8.7% (16)
Socket interface	14.0%	13.9%
VMTP	25.0%	23.6%
Sleep, Wakeup	4.1%	4.5%
IP time	7.4%	6.6%
Software int	1.3%	0.5%
Others	2.1%	2.2%
Unaccounted	2.7%	0.7%
Total	98.5%	98.6%

Figure 2.11: Components of server-side processing for small data exchange in a VMTP implementation on Unix

Part	Sun3/50 Client	Sun3/160 Server
ByteCopy	24.5%	14.61%
Checksum	13.59%	9.03%
Data buffer manipulation	6.38%	7.65%
VMTP	12.61%	13.22%
IP time	11.11%	5.22%
IPL	9.68%	9.46%
Software int	3.45%	0.18%
Others	1.5%	3.29%
Total	95.25%	95.14%

Figure 2.12: Components of packet processing for a 16-Kbyte read in a VMTP implementation on Unix

Part	VAX
User-kernel move	13%
Checksum	12%
Device DMA	20%
TCP/IP	16%
Others	30%
Unaccounted	8%
Total	1525 ms

Figure 2.13: Components of packet processing for large data transfers in a TCP/IP implementation on Unix

The measurement environment consists of an implementation of LLC for a single station where service access points are connected to a message source/sink and where the network is emulated in software and is assumed to be of infinite capacity. Performance measurements for virtual connection traffic are made on an active loop-back connection open between two service access points of the station; similarly, the measurements for datagram traffic are made on datagrams transmitted between two service access points of the station. In these measurements, a message is always waiting to be transmitted and receive buffers are always available; thus, the LLC program is always either sending or receiving a message. Packets are transmitted (and received back at the node) without any delay or transmission errors. The environment described above, thus, appropriately focuses the attention on the internal processing done within the LLC protocol.

The program implementing the protocol is written in C, and can run on a network interface using a general-purpose microprocessor. Since the program allocates its own resources and schedules various asynchronous activities, it can run without the support of a general-purpose operating system. However, to facilitate measurements in this study, we ran it on a VAX 11/780 processor under UNIX (a general-purpose operating system) and used the program monitoring tools available on UNIX, namely *GPROF* and *MONITOR*. An execution time profile generated using these tools consists of absolute and relative execution times of functions called during the execution of a program.

The program consists primarily of four major modules: *GETCOMMAND*, *SEND*, *RECEIVE*, and *GIVERESPONSE*. These modules are called in a round-robin order by a scheduler named *ROBIN*. *GETCOMMAND* processes commands received from a higher layer. Among the commands processed are commands to open and close virtual circuits, and commands to send and receive data. When a data message is to be sent on the network, *GETCOMMAND* packetizes the message and the packets thus formed are queued in a transmit queue reserved for the circuit to which the data message belongs. *SEND* polls each virtual circuit in a round-robin order, and transmits the waiting packets with the appropriate headers. The transmission of a packet is effected, in this case, by handing it over to the network emulator. *RECEIVE* processes each packet received from the network according to its type, the state of the virtual connection, and the state of the destination SAP. If the received packet contains data, *RECEIVE* locates a message buffer, copies the data into the buffer, sends an acknowledgment packet, and forwards the filled message buffer to a higher layer protocol task. *GIVERESPONSE* dispatches responses for the commands received to the appropriate higher layer entity. Datagram traffic is handled in a way similar to above with the exception that, *RECEIVE* does not generate an acknowledgment packet for a packet received as a datagram.

The three functions that deserve special mention are the ones which together take up a large fraction of the total processing time. One of these is *BMOVE*, a function for block movement of data within the memory. *BMOVE* is called every time a message buffer is packetized, and the data from a received packet is depacketized to form a message. For the environment described above, this is the function required to copy data in and out of the network interface, since typically a user message exists in the memory of a host and packet buffers are located in a network interface. The other two functions, *ENQUEUE* and *DEQUEUE*, are primitives for accessing queues. These three are simple functions which require no more than four or five source statements (in C language). The remaining functions are not described here, as their names suggest the tasks being performed.

Preliminary results showed that *BMOVE* takes up a large fraction of the total execution time. This led to the development of three distinct versions of the program, all of which differ by the way the block movement of data is done. In the original program, referred to as Program A, *BMOVE* was coded as a

single statement in C language. For this program, it was observed that of the total program execution time for sending a 1500 byte long message on a virtual connection, more than 90% was spent in *BMOVE*. This observation led us to develop two improved versions of the original program. In one version of the program, referred to as Program B, the function is made more efficient by utilizing a block movement instruction available on the VAX machine. In the other version of the original program, referred to as Program C, *BMOVE* sets up a DMA channel to transfer data, but the function itself does not move any data. The assumption made in this version is that the hardware architecture for the network interface includes DMA channels, and the latter are able to transfer data to/from a packet as fast as it takes the processor to process a packet. In the following, we discuss the results of measurements of these three programs for virtual connection traffic traffic.

The first set of experiments constituted moving a number of data messages, (specifically 1000), over a loop-back connection. The size of a data message was arbitrarily chosen as 1500 bytes, and this was also selected as the maximum size of data in a packet.[2] The difference in performance amongst the three programs is shown in Figure 2.14, which contains the total program execution time and the relative execution time of *BMOVE* for sending a single message over the connection. The effective end-to-end throughput is calculated over the time taken to transmit 1000 messages. The end-to-end throughput for program A is 375 KBPS, which is comparable to that observed with other commercially available LAN interfaces. The end-to-end throughput for Program B and for Program C is 7 times and 15 times better than that of Program A, respectively. We note here that since the same processor is used to send and to receive data in a loop-back connection, the end-to-end throughput for a loop-back connection is half that available for a connection whose end-points are on different processors.

The execution time profile of Program A is uninteresting because only one function dominates the program execution time. Figure 2.15 shows the execution time profiles per message for Programs B and C. In Program B, *BMOVE* takes up 41% of the total execution time; whereas, in Program C, it takes only 10%. The execution profile of Program C also shows

[2] The maximum size of a packet for 10 MBPS Ethernet is 1536 bytes.

	Total Program Execution Time (ms)	Execution Time of Bmove Function (%)
Program A	32.15	91
Program B	4.56	40
Program C	2.19	10

Figure 2.14: Relative execution times of block movement of data

that the queue management functions, namely *ENQUEUE* and *DEQUEUE*, now constitute the major fraction (about 40%) of the total execution time.

A major observation that can be made from the results obtained pertains to the design of the adapter's system architecture. In general, the processing of packets can be separated in two parts; processing of a packet header, which we call packet-level processing, and processing of each data byte, which we call byte-level processing. The checking for the correct sequencing of packets is an example of packet-level processing; whereas, the copying of data in a packet, i.e., packetization, is an example of byte-level processing. Other examples of byte-level processing are error detection and correction, bit-order or byte-order reversals, and encryption and decryption. In Program C, the packetization and the depacketization functions are separated from the packet-level processing and are performed concurrently in hardware. In general, the separation of packet-level and byte-level functions is advantageous because one is able to design systems which handle both of these functions concurrently and with matching speeds. Moreover, the byte-level processing is generally simpler and repeated more often; thus, it is cost-effective to design fast hardware specifically to perform byte-level processing.

2.7 The Common Case

In our design, we concentrated hardware and software resources available for efficient execution of the common case in packet processing. To determine the common case in our environment, we measured the frequency of occurrence for packet loss, retransmission and out-of-sequence receipt of packets. To measure these parameters, we recorded network traffic

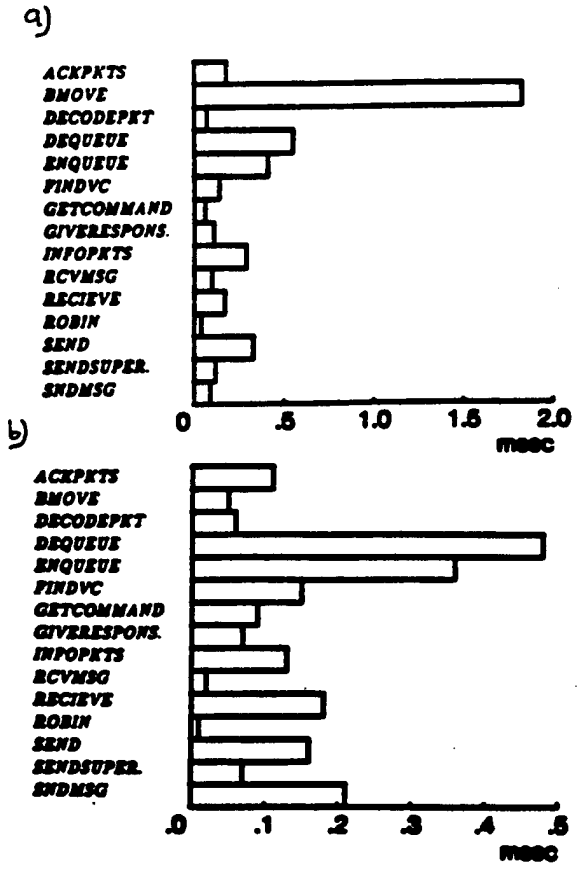

Figure 2.15: Profiles of the program execution time of sending a data message (1500 bytes) on an already open virtual circuit using a) Program B, and b) Program C.

at two separate workstations for over a period of 72 hours. On a Sun 3/50 workstation, we observed over a million packets. Of these, less than 0.05% were received with wrong CRC or with packet overrun indications. At a VAXstation II, we observed about 820,000 packets of which roughly 0.1% were received in error. The VAXstation II is roughly 3 times slower than the other workstations studied, which perhaps accounts for the slight increase in packets received with error. We also found that packet retransmission occurs with less than 6% of the total packets transmitted, and very few of them are retransmitted more than once. Less than 2% of the multi-packet transactions resulted in out-of-order packets. Moreover, we found that when a packet retransmission occur, it is more often due to delayed response rather than a lost packet.

These observations suggests that in a typical LAN environment the common case is a valid packet received in right sequence without a transmission error. It is instructive to see how the processing time is divided among functions performed for the common case. To do this, we traced the code written for VMTP under V and counted instructions executed for the common case. The VMTP header processing for the common case takes approximately 40 instructions. The total instructions needed for transmission of a maximum size packet are 1200 instructions for the first packet of a packet group and 850 instructions for the subsequent packets. The data movement and checksumming functions require about 14 instructions per data word. The surprising result of this measurement is to realize how few instructions executed per packet depend on the state machine of VMTP protocol.

2.8 Complexity of Transport Protocols

Complexity of state machines of conventional transport protocols does not contribute much to the cost of packet processing. This fact, supported by measurements, implies that simplification of state machines is not the most rewarding approach for increasing transport-level performance. In case of VMTP, as observed in the previous section, header processing takes less than 40 machine instructions out of 850-1200 instructions per packet. A similar observation was made for TCP, a standard transport protocol, by

Error-Control Algorithm	Processing Time Per Packet (μs)
Selective Retransmission	56
Sliding Window (TCP)	40
Go-back-N	45

Figure 2.16: Processing times for alternative algorithms for error and flow controls

David Clark [17] and Van Jacobsen [37]. Both studies show that fewer than 50 instructions are necessary to process a TCP packet header.

Further evidence arguing against simplification of a state machine is also presented by the comparison of the performance of UDP with that of TCP or VMTP, shown in 2.4 for their implementations under the Unix operating system. Although UDP is a "simple" transport protocol - does not provide reliable transfer of data - its performance is not significantly higher than complex protocols such as TCP or VMTP.

Would we significantly change the makeup of the packet processing cost by using different mechanisms for transport-level error control and flow control? The answer is no as evident by the comparison of computational requirements of various algorithms implementing transport-level error and flow control policy. In Figure 2.16, we show the computational requirements per packet for selective retransmission, Sliding-window, and Go-back-N policies. These were calculated from the best published code for each of these mechanisms executed on a Sun 3/50 workstation. Note that these should not be incorrectly interpreted to mean that throughput observed in each case is identical. The throughput observed depends on the environmental factors such as probability of lost packets, signal propagation time, and network latency.

2.9 Results

Measurements shown in this chapter support the following conclusions.

- ***Minimizing the cost of checksumming, encryption and copying of data would significantly increase throughput for large data transfers.***

  In a typical implementation, the time required to execute these operations is roughly 50% of the total processing time required per packet. Hence, reducing this cost should significantly increase performance.

- *Simplification of state machines of conventional protocols is not required for increasing transport-level performance.*

  State machines of conventional protocols specify how to process a packet header. The typical cost of header processing was measured here to be less than 10%-15% percent of the cost of packet processing. Although simplification may bring down the cost of header processing, it will not significantly increase performance, until the other costs in packet processing are reduced first. One of the other important cost to reduce is the cost of data-related operations, as mentioned above. The other important cost to reduce is the processing overhead in a device driver and in a operating system.

- *Currently, the performance available to a user is limited by a host interface.*

  Increasing speed of networks from current 10 Mbits/sec to a gigabit/sec range does not promise to increase user-level performance, unless host interfaces also improve. For a packet of the maximum size allowed by Ethernet, we measured that the packet transmission time is negligible as compared to current packet processing times at end-nodes. Increasing the maximum size of a packet, which would be feasible in high speed networks, does not affect the validity of this assertion. As one increases packet size, so does one increase the packet processing time, which includes overheads such as checksum and data copying. Indeed, we find that the increase is proportionately much larger in the packet processing time as compared to that in the packet transmission time.

Chapter 3

Host Interface Architecture: Key Principles

3.1 Introduction

In this chapter, we address the central question of how much intelligence should an adapter have. Most current adapters have no onboard intelligence to perform transport-level functions. We term these adapters as "dumb adapters". We use the term "intelligent" adapter for those that perform at least some transport-level functions. In the published literature, we find proposals for intelligent adapters ranging from hardware implementation of an entire protocol [13, 32, 46] to those that provide hardware support to few critical functions [15, 30, 39, 42, 51]. None have addressed the question of how much onboard intelligence is necessary or sufficient for high performance.

3.2 Onboard Intelligence

We argue that the adapter should perform transport-level function such as

- Transport-level checksumming
- Encryption and decryption
- Header processing for intermediate packets

This choice, as we show below, minimizes data movement over the host bus and memory and reduces per packet processing time. An additional benefit of having onboard intelligence is that one can protect host against network malfunctions. These issues are discussed next.

3.2.1 Minimization of Data Movement

Ideally, the network-bound data should be transferred just once on the host bus. Currently, additional movement of data is required due to checksumming, encryption and copying of data involved in forming packets. Checksumming requires that each word be read once; encryption (and decryption) requires that each word be read and written once. Performing these functions by a host processor implies that each data word be moved at least twice over the host bus; read (and process) a data word and transfer it to a network device. An additional bus transfer per data word is necessary whenever the host either uses the DMA mode for transferring data to a network device or accesses network-bound data through a cache that uses a write-through mechanism. This situation is illustrated in Figure 3.1 (a). Additional use of bus capacity is for moving instructions for processing packets. The overhead of instruction movement is minimized by using a processor cache, but caches presents another set of problems in handling network traffic. Network-bound data increases traffic through a cache and also causes cache pollution, since network-bound data is fetched and used only once. Cache pollution results in increasing cache miss ratios and thus again decreasing performance.[1]

[1] In order to quantify the effect of cache pollution on cache misses, we ran an experiment on a multiprocessor system with per-processor caches. In one experiment, we observed a 6% decrease in the number of cache misses with a modified driver (and cache management software) that used only fixed memory locations to transfer network-bound data. The application ran was formatting of a 30-page document, using "latex" package and the system was cold-started and loaded with same number

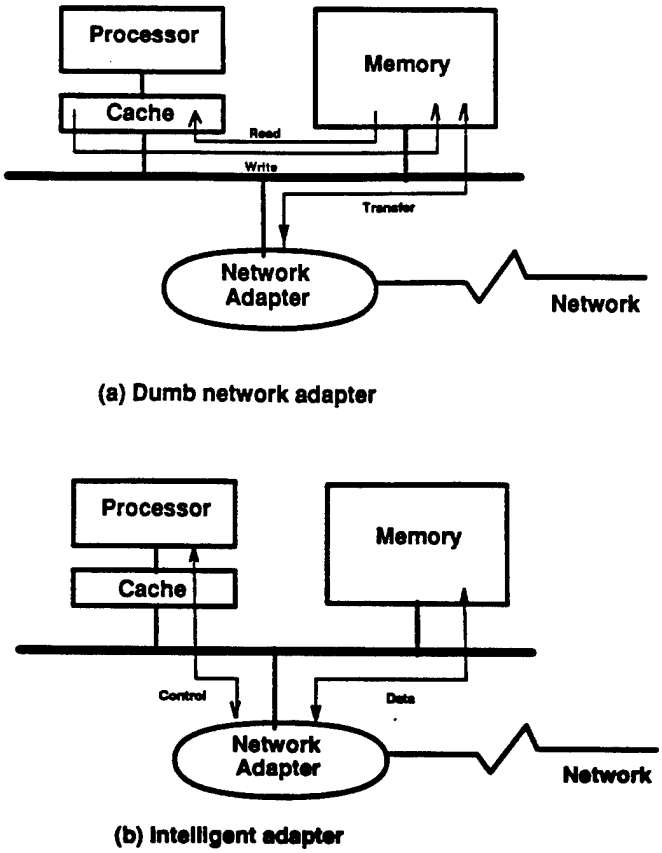

Figure 3.1: Data Movement with (a) a dumb adapter and (b) an intelligent adapter

The best solution for minimal data movement is to use adapter to perform checksumming, encryption and packetizing. In this case, shown in Figure 3.1 (b), each data word moves exactly once over the host bus. Additional benefits are that the cache traffic is reduced and the cache pollution is avoided. Network-bound data and code for packet processing bypass processor and its cache.

The second-best solution would be to use a software-controlled cache [10]. This type of a cache can be programmed to minimize pollution of cache by network-bound data. One avoids cluttering a cache by reusing the same cache locations to read in the network bound data. Furthermore, the data may be directly copied into the device memory, eliminating a bus transfer. An alternative scheme is to allow network device to read data directly from a cache memory. For this alternative to work optimally, one also should ensure that cache does not automatically write-out dirty pages to main memory. Although both schemes minimize memory references per data word and avoid cache pollution, these still require 2 bus transfers per data word.

Current interfaces copy data between user's and kernel's address space and sometimes within the kernel's address space. Consider an example of a distributed file server recently implemented by Garret [60]. The typical path of data movement in a read operation for the file server are illustrated In Figure 3.2, we show how often data moves within the system in order to perform a single read operation for this file server. The example shows that data is copied seven times before becoming available at the destination. The first copy occurs when data is moved from user address space to kernel address space. Then, it is moved to form a packet, breaking up the message if necessary. Finally, the packet buffer is copied to adapter memory. A similar set of operations are made at the receiving end. In some systems, copying of data between users' and kernel's address space may be avoided by remapping physical pages containing data to a different virtual address space. But, this operation of remapping of pages is costly enough in most systems to motivate designing adapters that avoid physical or virtual copying of data.

Copying data can be avoided by using intelligent adapters in the following way. The scheme is elegant and simple. The operating system of a host reserves buffer space in a user's address space and passes along buffer pointer(s) to the adapter. The adapter will use these pointers to form packets, and to transfer data directly to these buffers, thus avoiding data transfers through the kernel. This model of interactions is termed here as the "buffer passing and reservation" model. In this scheme the device and the host interacts only once per message transmitted or received. At the sender, the message, which may be in non-contiguous physical pages mapped into the user's virtual address space, is locked by the operating system, and the page pointers are passed to the device along with the information necessary to form packets out of it. At the receiver, the destination process reserves buffer space, which is locked in by the operating system, and their pointers are provided to the adapter along with filtering information necessary to identify packets received for this destination pro-

of applications in the same sequence in each case.

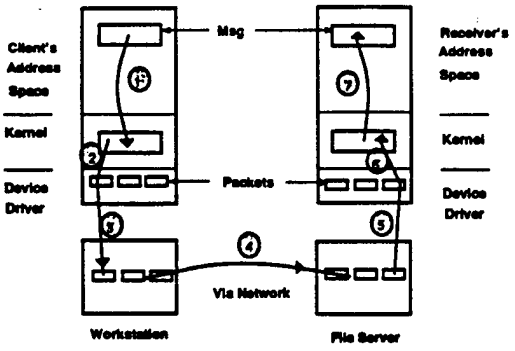

Figure 3.2: Data copies in a typical distributed file system

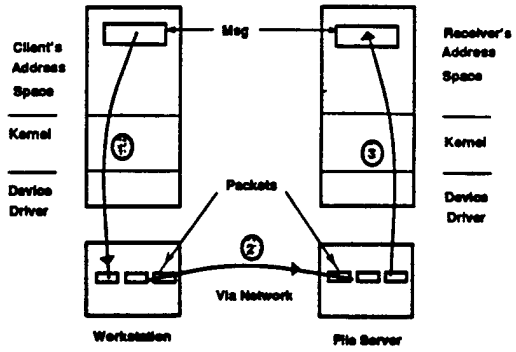

Figure 3.3: Data copies in a distributed file system using the NAB

cess. We refer to this model as the buffer passing and reservation model for host-adapter interactions. The details of messages sent and responses received across this interface are described in the next chapter.

This model avoids a need to make either virtual or physical copy of data from user to kernel space and vice versa. Only one copy of data is made in this model, which is needed anyway to transfer data to/from network. A distributed file server built with this interface is shown in Figure 3.3.

A technique called Scatter-Gather DMA (SG-DMA) technique is often used to minimize data movement. The technique most easily implemented at a sender. At a sender, one can form packet headers in kernel space and program the SG-DMA device so as to link it with the data contained in the user space to form a complete packet. This is the gather part of SG-DMA which does eliminate a copy of data from user to kernel memory space. The problem is at a receiver, which implements the scatter part. The scattering refers to the mechanism whereby received packets are split into header and data parts that are stored separately in the host memory. Without doing any transport-level processing, the adapter at this point is unable to directly move the data part to its ultimate destination, thus failing to eliminate kernel to user data move. A statistical technique such as optimistic prediction of host memory location, as proposed by Carter and Zwaenepoel [4], succeeds in avoiding user-kernel move typically 50% of the time. When the prediction fails, one requires additional processing and data movement to correct the mistake.

Additional optimization is feasible for network exchanges involving small amounts of data such as making a remote procedure call. The reason is that, typically, such data is recently referenced and hence is in cache prior to its being sent on the network. Thus, to pass a pointer as is done for large amounts of data would require that the data in cache be first written back to the main memory. Moreover, the adapter spends additional time getting data and processing to form the packet. This additional time is all the more significant as latency is the important measure of performance for this type of data transfer. Since such exchanges are frequent in a fully distributed operating environment, it is an important optimization to make.

The procedure is simple. Either a user or the operating system may flag transfers involving only few words. When that occurs, the host includes data along with the other control information necessary to form a packet. The data and control information is arranged in the same format as a packet, thus the adapter needs to do no more work to send it out on the network. The procedure reduces latency by avoid-

ing explicit write-back for cache data and by simplifying the processing of such packets at the adapter.

3.2.2 Reduction of Packet Processing Time

Current intelligent adapters such as [18, 23] execute transport protocols with lower performance compared that available with a dumb adapter. This was offered as one of the main arguments against using onboard intelligence. Our investigations suggest that intelligent adapter can do much better by successfully utilizing three general techniques for reducing packet processing time. The first technique is to overlap the header processing of a packet with the next packet's transfer to host and network. Since current memory and bus architectures of adapter limit improvements brought by the overlapping, we propose here a novel memory and bus architecture that make the overlapping more effective. The second technique is to use the pipelined processing for a packet. The third technique to use is header prediction reducing protocol-specific processing of packet headers. We discuss these techniques in this section.

Adapter's Memory and Bus Architecture

Currently, concurrent activities such as processing of packets with the reception and transmission of packets, and data transfers from the host slow down the packet processing at the adapter. The bottleneck is the adapter's memory and bus. The adapter memory is used for everything: as a staging area for network-bound data, as a transient storage for received packets and for programs used to process packets. Thus, the total memory references made to the adapter's memory are comparatively larger than either those made to main memory or host processor caches. Thus, the conventional memory architectures used in current adapters leads to low performance. Moreover, the future generation of fast microprocessors with 10-100 ns instruction cycle times [29] will require a high memory bandwidth to run at full speed. Thus, bus cycles used in arbitration and data transfer will further slow the effective rate of packet processing.

To solve the problem, one should design high-speed memory architectures. We proposed a novel, cost-effective architecture based on Video-RAM components [39]. The basic component, i.e. VRAM IC, has two independent access ports, one is connected to a randomly-accessed memory cell array and the other is connected to a serially-accessed, large shift register [47]. A wide data path connects these memories. Transfer of data between them takes place in a single access cycle. With this we have provided an architecture where data movement happens independent of packet processing. An entire packet received or data transferred from host memory is accumulated in the serial memory before being transferred to the random-access memory for transport-level processing by the adapter's processor. With such an architecture, we are able to eliminate bus interference[2] between data transfers through the adapter and processing done at the adapter. Data transfer rates are higher than that obtained with regular memory components since serial memory is faster. The higher speed is a result of the fact that serial memory on-chip has lower complexity than the memory cell array of DRAMs. In commercially available VRAM parts, the serial port has at least 5 times faster transfer rates than either its random port or DRAMs using a comparable process technology. For the parts we have used in our prototype, with 32-bit wide memory, we achieved a transfer rate as high as 800 Mbits/s.

The data transfer on the host bus can also be made efficient by using block transfer bus protocols offered by bus standards such as VMEbus, MultibusII, and Futurebus [3, 26, 33]. These protocols transfer data at a higher rate because no address set-up is needed except at the beginning of a block transfer. A typical DMA device transfer requires address set-up for every data word transferred, which limits the transfer rate for a block transfer of data. For instance, for a VME bus, peak transfer rate of 320 Mbits/s and average transfer rate of 220 Mbits/s are feasible as compared to a maximum 40 Mbits/s available for a normal DMA transfer.

Pipelined Processing

In conventional transport protocols, there is little parallelism of the type that allows one to use parallel processing to reduce per packet processing time [41]. On the other hand, the pipelining, a form of parallelism, seems natural for this purpose. For example, consider an architecture where functions such as checksumming and encryption are performed on data as packets are being pulled out of memory or

[2] Each data block transferred requires only two memory cycles per packet transferred, an overhead of less than 1% for a minimum size of packet transferred.

put into memory at the network end. Each data word passes through the pipeline stages that perform simple operations such as addition and rotation as defined by checksumming and encryption algorithms.[3] These stages could be implemented with hard-wired controls and performed at the same rate as the network up to the gigabit range. Let us define the depth of this pipeline as the time needed to pass a single word through it. The latency for small data exchanges is kept low by minimizing storage of data within the pipeline, i.e., reducing its depth. In general, the pipeline needs no more storage than what is required for executing checksumming and encryption functions. It never stores an entire packet. We designed and implemented this pipeline as a part of the intelligent adapter [39]. The prototype required no more than 10 words; thus, its depth is less than 10% of the total transmission time of the minimum size VMTP packet.

The main advantage of the pipeline is that it reduces memory references made to the adapter memory. The references required for instruction and data fetching to perform checksumming and encryption are eliminated. This cost is folded with the cost of moving data in and out of adapter memory to and from the network link. Because of reduced memory references and the shallow depth of pipeline, per packet processing time is also smaller than the case where the pipeline is not used.

Header Predictions

The one remaining part of the packet processing that we have not yet considered is protocol-specific processing of packet headers. In this section, we will discuss header processing algorithms based on header prediction techniques. The algorithms offer considerable improvement over the ones in current use.

At the sender, we can reduce processing time by noticing that subsequent packets formed out of a message have similar headers. By reusing the same packet buffer and making appropriate changes in the header portion, one will minimize processing required to form subsequent packets in a packet group.

At the receiver, the processing time is reduced by using prediction techniques that exploit the observed locality of network traffic. A definition of the principle of network locality is that the packet received next is frequently related to the one received last, i.e., what happened in the past is frequently a good indicator of the near future. This observation has been made by a number of studies [4, 25, 36, 38, 45, 62]. The factors that increase locality are decompositions of long messages into packets with similar headers, transmissions of multiple packets in a single access to the network, and generally low utilization of networks. The factors that decrease locality are multiplexing of traffic from other local processes or simultaneous demands by multiple clients to a single server.

One or more packet headers can be predicted and stored at a receiver to be used as a hint for processing a packet. A general outline of the algorithm is as follows. An incoming packet is compared to a stack of predicted packet headers. If a match occurs, little further processing is necessary, since the matching verifies that the packet is correct and also retrieves an index to the connection's control information including the host memory location to which data goes. The index and the predicted host memory address are stored in the stack. The stack is kept in the order of last accessed (or recently formed) header. The matching uses a simple linear search beginning with the last used (or formed) header. Replacement uses least recently used strategy. The hit results in about 1/4 of the packet processing time required otherwise. Thus, if prediction always succeeded we would have a speed-up of 4 times in packet processing rate.

The actual speed-up would depend on how often the prediction succeeds. To study this issue, we collected a number of traces of VMTP traffic on our 10-MB Ethernet. A packet trace is collected with an modified kernel that saves in memory all the key fields of packets received by the host. We collected traces for a workstation and a file server, both Sun 3/75 machines. At the time of collection, there were at least 20 machines intermittently accessing a file server machine. and the workstation under study was being used to compile the V kernel, to run a text formatter program on a long paper, and to read my mail and to edit a small file, each running in a different window. Typical length of each trace is about half an hour long, collecting exactly 15,000 VMTP packets per trace. During this time, most of the programs started at a client workstation completed. Since the temporal information, i.e. inter-packet arrival times, is not important in this study, we do not expect the results to

[3] Strictly speaking, one also needs hardware to determine when to start and stop checksumming or encryption, since only transport-level packet is checksummed or encrypted. Since VMTP packets have fixed header and length fields, hardware implementation of this part is easy.

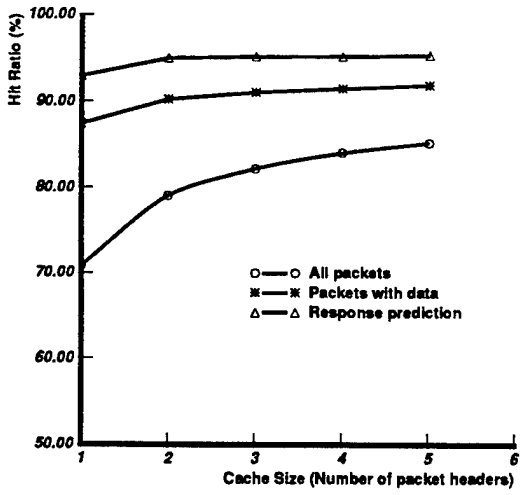

Figure 3.4: Locality of network traffic observed at a workstation

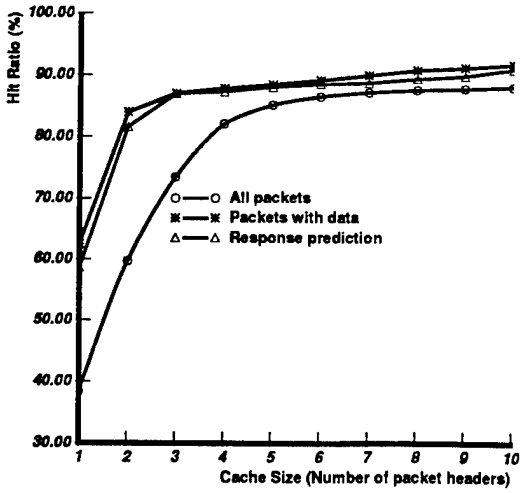

Figure 3.5: Locality of network traffic observed at a file server

be affected by the modifications made for statistics collection, which slightly increased per packet processing time.

We define the hit ratio as the number of times match occurs divided by packets received. The cache size is defined as the number of predicted headers stored. In Figures 3.4 and 3.5, the hit ratio for three different prediction algorithms is shown as it varies with the cache size. The first algorithm based its predictions on all received packets. The second one based its predictions from the packets containing full data size. The third one based its prediction on the packets with full data size plus response packet headers derived by observing outgoing request packets. We refer to these three algorithms as naive, data-based, and response prediction algorithms, respectively. Figure 3.4 and 3.5 show results from traces collected on a client workstation and a fileserver node, respectively.

The hit ratio increases asymptotically with the increase in stack size. The asymptotic value is reached for stack sizes 5-6 for a client workstation. Our collected data shows (not plotted in these figures) that to reach 98% percent hit ratio requires a rather large cache size (125). Since the cost of matching algorithm, for success as well as failure, increases linearly with the stack size, the stack size has to be limited to a low value. The data-based and the response prediction algorithm both worked better than the naive algorithm in all cases. The response prediction was observed to be much better than both of the others for the client workstation. Its ratio improved very rapidly with the stack size, requiring no more than 2 entries to reach a plateau. The data-based prediction and response prediction worked identically for the server. This is explained by the observed fact that a file server generated few request packets. The plateau in hit ratio was reached for cache size of 6-8 entries for a server as opposed to 3-4 entries for a client. The difference exists because of the server receiving simultaneous traffic from multiple clients.

3.2.3 Overload Protection

An additional benefit of having onboard intelligence is that hosts can be well-protected against network malfunctions or hostile users. Currently, most hosts stop doing any useful work when the traffic from network exceeds its capacity to handle it. The primary reason is that a host gets involved in processing packet which are dropped at transport or higher layers. This allows an opportunity to a hostile remote user or some kinds of network malfunction to pose a threat to a host. The term "overload protection" is used here to refer to mechanisms designed to protect a host from such traffic.

The buffer reservation model of a host-adapter interface, suggested in Section 3.2.1, offers the necessary mechanism for overload protection. Any packet that does not have a waiting buffer reservation from its destination process is dropped on the floor. Instead of notifying a host for every dropped packet, the adapter summarizes this information and periodically sends it to a host. The period at which such notifications are given is programmable by host software. Further protection is provided by allowing host to suppress traffic with a general filtering function sent to the adapter. With it, an adapter can prevent packet traffic to specific destinations, from specific sources, types of packets, and any combinations of these conditions.

This scheme offers a better protection than that offered by disabling of packet interrupts. A host can disable packet interrupts preventing any packets from being received, thus protecting itself against the traffic overload. Unfortunately, disabling of interrupts would discard legitimate packets sent to the host, thus fulfilling in part the intent of the hostile user. Whereas, the powerful and general mechanism offered by the buffer reservation model, coupled with the adapter fast enough to receive and throw out packets at the maximum rate of the network link, may allow a host to continue to receive and to send packets even with the overloaded network. Note that this mechanism provides only one-way protection. To protect network from being flooded by a host, one should add mechanisms to detect hosts currently sending excessive amount of traffic and mechanisms for isolating offending hosts from the rest of the world.

3.3 Alternatives

In this section we will examine proposals for host interface architecture that do not use onboard intelligence. The primary reason for considering them is the cost of intelligent adapters. One of the proposals is to increase performance by reducing number of packets per data transfer. The second proposal is to avoid storing packets fully at the adapter. The third one is really an objection often raised against intelligent adapters.

3.3.1 Larger Packet Size

Increasing the size of packet reduces the average number of packets formed for exchanging user-level data. This has been argued by some researchers as the right approach for increasing performance. The argument proceeds like this: Since the performance is limited by packet-processing time, reducing number of packets per segment by using larger packet sizes, should increase throughput. As we show in this section, the argument is superficial and is based on the experience of current host interfaces.

The throughput thus increased by increasing packet size has an upper limit that is determined by the make-up of packet processing time and the host interface. A cost of processing packet has two parts: the fixed part, due to processing of headers, and the variable part, due to operations such as copying and checksumming of data. In current interfaces [1, 23, 59], the fixed part dominates the variable part. Thus, the performance improves as the packet size increases. But with a packet that is say 10 times the size of the biggest Ethernet packet, the variable cost dominates the fixed part. At this point, further increase yields marginal increase in throughput. Additionally, in NAB interface architecture, the fixed processing overlaps with the variable part. In such a case, increasing a packet size beyond some size would not increase throughput.

Larger maximum size of packets causes three major problems. In network transport involving hops through store-and-forward routers, larger packet sizes increase end-to-end delay. The larger packet size also increases probability of a packet being lost due to error. The other problem is increasing the response time for other users, because increasing maximum packet size results in disproportionately higher delay for users with small amounts of data to exchange.

3.3.2 Cut-through Adapters

We define adapters that do not need to store a full packet while processing as "cut-through" adapters. These have been considered "ideal" interfaces by some researchers [61]. Our analysis shows that cut-through adapter (CTA) is either marginally better or much worse than the intelligent adapter (IA) depending on the level of other bus traffic on host bus.

We used a simulation to compare their performances. The CTA is modeled as a server that is faster than the maximum arrival rate of packets. The packet is put into a FIFO waiting for bus access. The size of a FIFO is smaller than the maximum packet size of the network. Bus access is modeled as

a random event with exponential distribution of time between two consecutive accesses. The distribution depends on the interference of bus traffic originating from other sources such as disk-to-memory and cache traffic. After the bus is acquired, data from FIFO is transferred in a burst of fixed size. The bus is not requested until FIFO collects data equal to this burst size. We studied response times when burst sizes are 32, 1024 or 4096 bytes of data. The maximum size of packet is assumed to be 4 Kbytes of data plus the size of a VMTP header. The IA transfers data over host bus only after a full packet is stored and processed. Figure 3.6 indicates the impact observed on response time for reading a 16 Kbyte data using two different types of adapters. Neither of them lost or retried any packet, since in absence of interference the bus is always available in time. The response time in IA is no worse than 7% of that of CTA up to a gigabit network speed. Intuitively, this is a result of IA overlapping packet processing with that of sending data. The time added due to the store-and-forward nature of IA is for the first packet. Figure 3.7 shows how much worse it would get for the cut-through adapter when bus interference is accounted for. Bus interference at the sender forces transmission of an aborted packet to be restarted. At the receiver, it will result in overrunning FIFO and thus result in a packet loss. For a 30% overhead, we observe in our simulation that the IA is 20% better than the CTA.

Additional problems argue against using cut-through adapters. The bandwidth of host bus is wasted when a packet is received with a transmission error. Moreover, a host is not protected against network malfunctions and hostile remote users. Without storing a packet fully it is not feasible to provide this protection. Additionally, implementations of fairness and priority mechanisms are difficult with cut-through adapters. With this interface, one is forced to service packets in a first-come-first-serve (FIFO) order. Any data including those which are controls for relieving network congestion would have to wait their turn.

3.3.3 End-to-end Checksums

This is not an alternative but really an objection raised against intelligent adapters. End-to-end transport-level checksums are calculated by an intelligent adapter. This means that data errors in adapter memory or transfer over bus would go undetected.

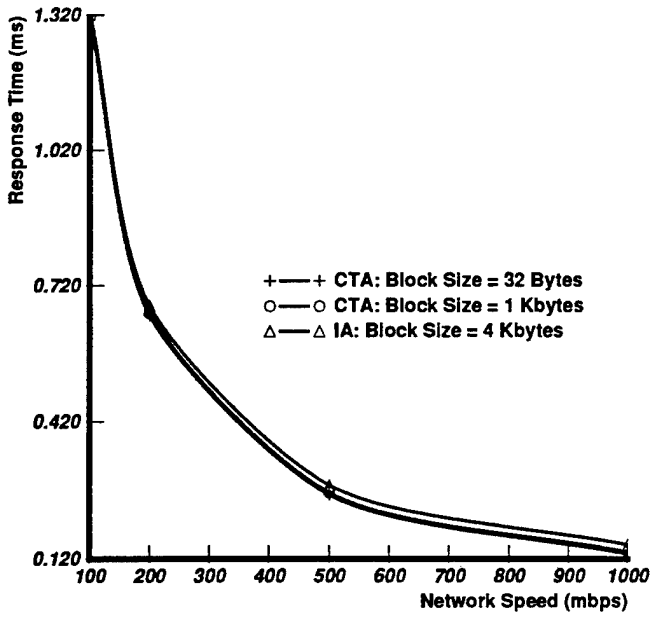

Figure 3.6: Comparison of an intelligent adapter with a cut-through adapter: No bus interference

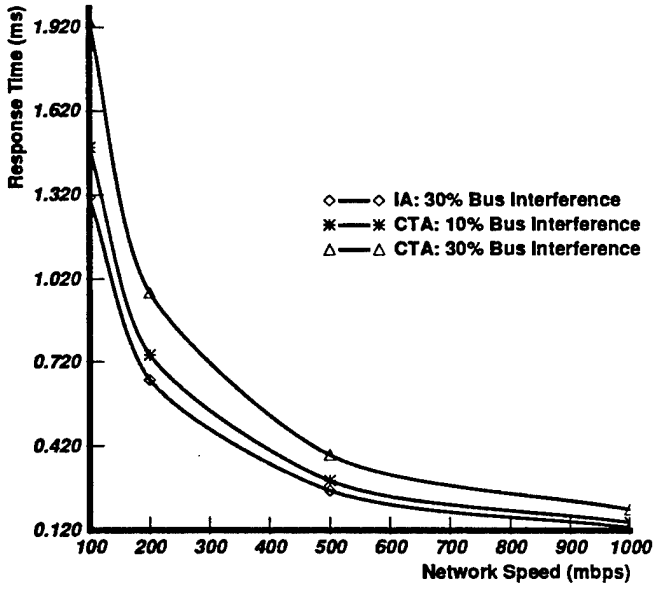

Figure 3.7: Comparison of an intelligent adapter with a cut-through adapter: With bus interference

This is perceived as a problem by some researchers in view of the end-to-end arguments made by [53]. We disagree. The real concern should be to ensure that a network link isn't the weakest link for the reliable operation of the system. Performing end-to-end checksums at the adapter is at least as safe as using the disk controller that does block-check errors and the memory controller that detects parity errors. For a typical computer system, traffic between disk or memory and host processor is carried by a host bus that does not protect against transmission errors. If one desires a more reliable system, one could use a parity line on the bus or block checksum hardware for each bus interface.

End-to-end checksums are performed at the adapter for the performance reason. Checksumming at a host affects the performance of network communications traffic and also wastes critical resources such as host bus and memory bandwidth. Thus, it affects the performance of the entire system. The alternative solution, checksumming at the adapter, results in a reliable system where network traffic is as well protected against bus and memory error as is the disk I/O or cache traffic.

3.4 Summary

In this chapter, we have addressed the question of how much onboard intelligence is necessary. We found that an adapter performing checksumming, encryption of data and processing of intermediate packets brings three major benefits. First of all, such an adapter minimizes data movement over the host bus to one per data word. The minimization saves host bus cycles, a critical resource, increasing performance of other users of the bus and also increasing performance in a multi-processor host. The second benefit is that the intelligent adapter with the "buffer passing and reservation" model, proposed here, can eliminate virtual or physical copying of data between user and kernel's address spaces. The third benefit is that the intelligent adapter saves main processor cycles, particularly important for hosts providing network services. Additional benefit of the onboard intelligence is that one can protect hosts better against network malfunctions and hostile users.

Current intelligent adapters are slow. This was a motivation for us to examine techniques for increasing their performances. We found three general techniques that are useful for reducing per packet processing time. Concurrent processing of packets with their transfer to network and host will increase throughput for multiple packets. The technique is particularly effective when the memory and bus architecture of the adapter is reorganized to permit true concurrency for these activities. The pipelined packet processing, proposed here, executes checksumming and encryption while packets are being transferred to and from the network. The pipelined processing folds the cost of checksumming and encryption with that of the transfer of packets and thus reduces per packet processing time. The third technique we studied is the prediction of headers of expected packets. We proposed a new header processing algorithm that used predicted packet headers as hints in processing received packets. The performance of the technique depends on how often the hints would become true. We analyzed packet traces collected in the operational environment and found that the prediction would succeed 65%-98% depending on the number of predictions stored and the prediction policy used. We also found that the prediction rate is consistently high for both client and server machines.

Finally, we compared the proposed choice regarding onboard intelligence with two alternatives: dumb adapters with large packet sizes and semi-intelligent adapters with minimal onboard storage of packets, also called cut-through adapters. The comparison with the dumb one with large packet sizes showed that the intelligent adapter has equal or better performance because of the overlapped header processing and data movement and has none of the disadvantages of the other such as high delay for exchanging small amounts of data. The comparison with the cut-through adapter showed that the intelligent adapter has almost equal or better performance than the other. For this comparison, we used a simulation that took into account the effect of interfering traffic on the host bus and delays in bus access. Results of the simulation measurements show that the intelligent adapter has about 20% lower request-response delay for a 16-Kbyte transfer with 30% bus capacity used for non-network traffic. For 0% bus interference, an environment that favors cut-through adapters, we found that intelligent adapters have less than 5% more request-response time compared to that of cut-through adapters.

In the next chapter, we describe a network adapter architecture embodying principles discussed here.

Chapter 4

NAB Architecture: Details

4.1 Introduction

Reducing packet processing overhead based on the principles discussed earlier depends on improvements in three separate areas: transport protocol design, the architecture of network device, interactions between host processor and network device. Previous experience suggests that improvements in any one do not significantly improve performance. We obtained dramatic improvements by making interdependent changes in all of these areas and proposed an architecture based on these changes. Our work in this area shows clearly the nature and scope of this interdependence. In this chapter, we describe only the relevant features of the proposed architecture, which we call Network Adapter Board (NAB) architecture. The details of the architecture are published in [39, 40].

4.2 Host-Adapter Interface

The host interface with the network adapter is designed to minimize the host overhead for network communication. The basic interface appears to the host software as a 1024-byte control register. To transmit data, the host software writes a control block, the *Transmit Authorization Record (TAR)*, to this control register. The TAR contains control information describing data to be sent including the pointer to data in physical memory. If the data fits entirely within the control register, the data segment description is omitted from TAR. In both cases, the network adapter transmits the data, as well as performs checksumming and encryption (if required).

For reception, the host software writes a control block, a *Receive Authorization Record (RAR)*, that "arms" the network interface to receive packets, specifying the maximum size to receive and the location in host memory into which to deliver the data. The RAR can specify any one of: (1) a specific source, (2) a class of allowable sources or, (3) any source for the received data. where source is either a transport-level or network-level address.

The interface interrupts the host when the received packet(s) satisfies one of these RARs, returning the RAR, along with the packet header, to the appropriate host via the control register. The returned RAR itself may contain small amounts of data in addition to the corresponding packet header. When the RAR is returned, the data has been already stored in the host memory at the location pointer(s) contained in it, unless the data is contained in the RAR. Incoming packets are discarded if they cannot be matched to an outstanding RAR.

The first byte in the control block distinguishes the types of the records passed via the control register. Four major types of records, used in transmission and reception of data, are an RAR with small amounts of data, an RAR with data descriptors, a TAR with small amounts of data and a TAR with data descriptors. Additional types of records are used by a host to add or delete acceptable destinations, to restrict traffic from a source, to provide decryption/encryption keys to the NAB, to get status information, and to reset the NAB.

The RARs and TARs include some common control information used by NAB, a packet header including a network-level header, and either small amounts of data or a list of data descriptors pointing to locations in the system memory. The common control information includes a link field, type of RAR matching, transport-level source and destination addresses, interrupt control flags, a local host number, and a local process identifier. The buffer descriptors are omitted for TARs and RARs containing small

amounts of data. A link field, used by the on-board processor, allows chaining of these records as necessary. The type of RAR matching to be done indicates if the RAR can be used for receiving traffic only from the specified source or any source. Interrupt control flags are used to determine when to interrupt the host identified in the record. On reception, one may have the host interrupted either when the data of the first packet is stored in the system memory or when the data is completely received or both. The RAR is written into the control register with the length of data received before the host indicated in RAR is interrupted. On transmission, one may have the host interrupted either when the NAB begins processing a TAR, or when the last of the data segment is transmitted. The TAR is written into the control register before a host is interrupted. The buffer descriptors in a TAR point to locations in the physical memory space where data to transmit is available. The buffer descriptors in an RAR point to locations in the physical memory space where the data is to be received. The returned RAR and TAR contain, in addition to the buffer descriptors, the number of data words actually received or transmitted.

In the following, we discuss how this interface efficiently handles small amounts of data, large amounts of data, and also allows the interface to act as a firewall, protecting the host from the network.

4.2.1 Short Message Handling

The latency with short messages is minimized because the short message is written to the interface as part of the TAR and read as part of the returned RAR on reception. The operating system interrupt handlers for the network adapter can directly copy the message data between the control register and the operating system data structures, moving the higher-layer data to its intended destination with minimal cost. Thus, the delay introduced in transmitting and receiving a packet with a small amount of data is no more than that incurred with a host directly handling the packet and using the interface as a staging area to send and receive packets.

Note that including the packet header in the TAR means that the processor writes a small amount of data to the interface for transmission yet the network adapter has minimal processing on the data to prepare packets for transmission. In particular, for small data appended to header, the network adapter need not do anything before starting network transmission, given that checksumming and encryption occur as part of transmission.

4.2.2 Long Message Handling

Host overhead is minimized for the transmission and reception of large amounts of data, typically in the range of 4-16 kilobytes, by passing descriptors rather than actual data. On transmission, the host writes one TAR and receives one completion interrupt, with the network adapter transferring the data from host memory with minimal bus overhead.[1] The network adapter handles the per-packet overhead of packetizing, checksumming, encryption and per-packet coordination. On reception, the host receives a single interrupt for each RAR returned after the data has been transferred into global memory. Again, the per-packet interrupt overhead is handled by the network adapter.

In sending and receiving groups of packets, the interface can afford to introduce some latency for the first packet of the group as long as the whole transmission and reception has less delay as compared to a host processor handling per packet processing. That is, for a small number of packets K, it should be the case that

$$K * P_{host} > D + K * P_{interface}$$

where writing the control record to the interface introduces a delay D in transmission over the host processor writing the data directly to the network, K is the number of packets to be transmitted, P_{host} is the time for the host to packetize and send one full-sized packet and $P_{interface}$ is the time for the interface to transmit one full-sized packet. The value of K for which this is true should be as small as possible, ideally 1 but certainly less than the common size of a multi-packet packet group.

4.2.3 Network Firewall

The interface architecture is designed to allow the network adapter to function as a *firewall*, protecting the host from network packet pollution, both accidental and malicious. In essence, a host incurs no overhead for network packets whose reception it has not

[1] The VMP memory supports block transfer using the VME serial bus protocol, thereby minimizing bus occupancy and arbitration overhead.

authorized; the interface discards all packets that are not compatible with an RAR provided by the host. As a particular example, if the host does not provide an RAR for broadcast packets, then garbage broadcast packets incur zero overhead on the host processor(s). In general, the authorization model of packet reception plus the speed of the network adapter insulates the host from packet pollution on the network.

4.3 NAB: Hardware

The network adapter internal architecture is designed to provide maximal performance between the host interface and the network architecture. The internal architecture is structured as five major components, interconnected as shown in Figure 4.1. These components serve the following functions:

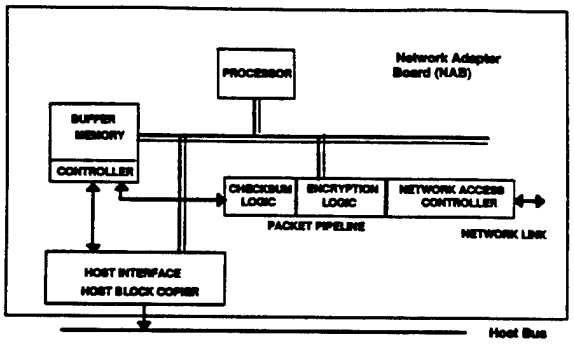

Figure 4.1: Network Adapter Board Architecture

Network access controller (NAC) : implements the network access protocol and transfers data between the network and the adapter.

Packet Pipeline : generates and checks transport-level checksums and performs encryption and decryption of data for secure communication.

Buffer memory : a staging and speed-matching area for data in transit between the packet processing pipeline and the host memory. Its specialized buffer memory permits fast block data transfers between the network and host and provides the on-board processor with contention-free memory access to the packet data.

Host block copier : moves data between the buffer memory and the host memory using a serial bus transfer protocol, minimizing the latency as well as the bus and memory overhead for transfers.

On-board processor : a general-purpose processor that manages the packet processing pipeline and various bookkeeping functions associated with the protocol.

Transmission is handled in three steps. When a Transmission Authorization Record is written to the interface, it is moved into the interface from the host memory by the host block copier. The TAR provides a description of the segment of data in the host memory to transmit. Next, the on-board processor forms the first packet from the TAR and first data blocks of the message and queues the packet for the packet pipeline using the information provided by the TAR. Finally, the packet is processed and transmitted by the packet processing pipeline and NAC at the network data rate. The pipeline calculates a checksum and optionally encrypts the data as the packet is transmitted.

On reception, a packet is accepted by the NAC and passed through the packet pipeline which decrypts the data, verifies the checksum, and deposits the received packet into buffer memory. If the received checksum is verified, the packet is matched to the appropriate RAR. For the first packet in a group of packets, this may involve locating and allocating a non-specific RAR, making it dedicated to receive more packets from this source. The packet data is then delivered to the host memory associated with this RAR. The reception of a packet into buffer memory proceeds concurrently with both the checksum verification and transfer to the host of previous packets. On reception of the final packet or on timeout, the host is interrupted and informed of the receipt of this packet group by returning the RAR in the interface control register. Thus, the host is interrupted only once per RAR used by the NAB.

If there is no matching RAR for a packet, the adapter determines whether the destination address is locally acceptable. An address is *locally acceptable* if the address is in the local host's list of destination addresses, both individual and group addresses. The adapter then transmits a response to the local host indicating that the packet was discarded. When the destination address is valid but no RAR is found, the interface discards the segment data and returns an indication to the destination host via the control register. The interface does not time out a partially-filled RAR; the partially-filled RAR is returned to the host only on an explicit request from a host. The host handles sending out retransmission requests and various timeouts. The host also sends acknowledgements and handles retransmission requests by issuing a new TAR.

Three key aspects to the design are the buffer memory, the packet processing pipeline, and the use of the general-purpose processor, as discussed in the following sections.

4.3.1 The Buffer Memory

The network adapter requires buffer memory in order to speed-match between the host bus and the network as well as to provide a staging area for the transmission and reception of packets. Three issues arise with the buffer memory design. First, the buffer memory must provide sufficient contention-free memory access to support simultaneous use by the on-board processor, NAC, and block copier, which happens under load. The performance of many so-called "smart" network adapters suffer from this contention. Second, the buffer memory must minimize latency for packet transmission and reception over direct transmission between the host memory and the network. Finally, a provision is required to prevent overcommitting the buffer memory to either transmission or reception, which would interfere with the functioning of the adapter. Our approach to each of these issues is discussed below.

Buffer Memory Contention

To minimize contention, the buffer memory uses dual-port static column RAM components, also referred to as Video RAM ICs. The Video RAM-based buffer memory, shown in Figure 4.2, provides multiple buffers to hold and to process packets while a packet is being received or being transmitted. This IC provides two independently accessed ports: one providing high-speed serial-mode transfer, and the other providing random-access. The serial-access port is used to move a packet from the network into the buffer memory or the data from host memory to the buffer memory. The random-access port provides memory access for on-board processing of packets by the adapter's general-purpose processor. The serial access does not need address set-up and decoding time so read-write times on this port are faster than read-write times for a RAM array. For instance, in our prototype, memory is 32 bits wide, the serial access time is 40 ns per word and the cycle time for random read/write access is 200 ns. This gives an effective transfer rate of 800 megabits/second over the serial port and 160 megabits/second over the random-access port. To provide the equivalent memory bandwidth on a single ported standard 32-bit wide memory would require a memory IC with read/write cycle time of 33 ns. Currently, such fast memories are available, but they cost more and have less memory density than video RAMs. Due to simultaneous memory accesses made by onboard processor

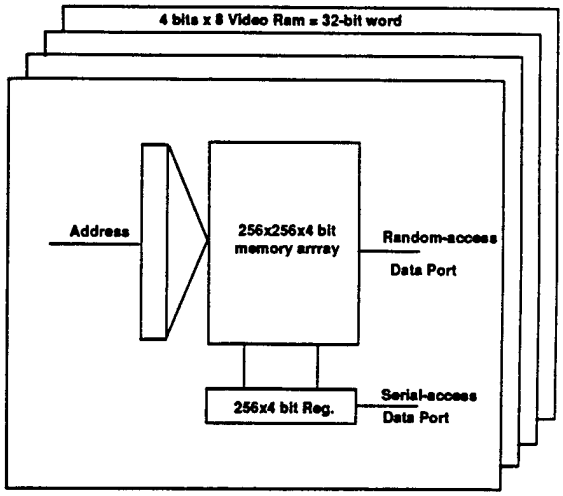

Figure 4.2: Video RAM-based Memory of NAB

and by network, page mode RAMs are less effective than Video RAMs in providing high-speed memory accesses. Interleaving of memory, which would reduce the buffer memory contention, was not the preferred solution because of its cost and because it would require faster memory-processor bus than that required in the VRAM-based memory.

Data transfer operations in this memory proceed as follows. A packet (or data) is received from the network (or the host) via the serial port and moved into the shift register contained in the Video RAM ICs. The shift register acts as the temporary storage. When the block is completely received, it is transferred from the shift register, in a single memory cycle, to a row of the memory cell array constituting the buffer memory. The processor manipulates header fields of packets stored in the array via the random-access port. The processing of a packet continues without interference while the next packet is being received, except for one memory cycle stolen for each received packet transferred to the memory cell array from the shift register.

Video RAM ICs provide performance that closely approximates the performance of true multiport memories, but at a fraction of the cost. A triple-port memory cell would triple the area of the memory cell, reducing memory density and increasing its access time. Video RAMs provide full memory bandwidth to the processor at low cost. They also allow high-speed block data transfer between the buffer memory and the host or network. The separate serial port avoids the processor losing memory bandwidth to arbitration overhead on the random access port. The serial transfer ports, accessed in parallel across a bank of video RAM ICs, maximizes the data unit to be transferred per arbitration between the host block copier and the NAC and minimizes the transfer time, thereby minimizing the arbitration penalty.

High speed FIFOs could be used as an alternative to the buffer memory described above to speed-match between the host and the network. Intuitively, a FIFO-based design should have minimal delay for two reasons. With short FIFOs, one begins to process and transmit the packet before the entire packet has been copied.

The FIFO approach also avoids the software overhead of managing input and output packet queues. Nevertheless, our buffer memory design provides better performance as we show in this section.

When the system bus is lightly loaded, our design compares favorably with the FIFO approach for large and small amounts of data transfers. For large data transfers, our design amortizes the buffering latency of the first packet over multiple packets. For a short single packet transfer, the difference in delay is small because the main source of delay in this case is the processing done at a host and the adapter, not the time of copying data to the interface.

However, at even moderate levels of bus traffic, our design outperforms the FIFO-based approach. When the bus is congested, the demand for bus access may not be satisfied in time to avoid underrunning or overrunning a FIFO, resulting in packet loss at a sender and at a receiver. The time thus lost in transmitting aborted packets as well as the retransmission delay increases the total time needed to transfer a data segment.

Mismatch between transmission data rates on the host bus and the network channel also supports using buffer memory. When host bus speed is much higher than network channel speed, the additional delay for bringing the first packet in full before beginning transmission is negligible compared to the total transmission time for a large data segment. When host bus speed is comparable to (or lower than) the network channel speed, bringing in a full packet before starting transmission is necessary to avoid (frequent) loss of packets by contention on the bus.

Buffer Latency

Buffer latency refers here to the additional delay caused by the on-board buffering of a packet as compared to the direct transmission by host to network. We minimize it by using a hardware block copier and by using contention-less memory accessing. A block copier transfers data between host memory and NAB using a serial memory transfer protocol, which minimizes transfer time and bus occupancy. Moreover, the NAB memory design described above facilitates high speed block data transfers [2] and provides contention-less memory accessing, both minimizing the buffering latency.

In this design, the cost of buffering latency of the first packet is amortized over the subsequent packets sent in the packet group. The subsequent packets are copied from the host memory or the network link

[2] The maximum transfer rate, using currently available Video RAMs, may be as high as 800 megabits/sec, assuming word width of 32 bits and 40 ns cycle time on the serial port.

in parallel with the processing of the previous packet transferred, hiding the cost of their buffering latency. For large amounts of data, the buffering latency of the first packet forms only a small fraction of the total delay. For instance, the minimum transmission time for 16 Kbytes of data over a 100 megabits/sec network is 1.31 ms; whereas, the buffering latency for the first packet in this packet group is approximately 25 microseconds, [3] which is less than 2 percent of the total transmission time. The latency is even less significant compared to request-response delay for large data transfers, as this delay includes processing times at the host processors.

We note that the data transfer between host and interface buffer memory does not constitute an extra copy because the NAB performs all the functions that the host processor would have otherwise performed if it was performing the transfer. That is, the interface memory is required as a staging area for incoming and outgoing network traffic in any case.

Reception of Garbage Packets

Ideally, the mechanism for matching a packet to an RAR is fast enough so that it cannot be overrun by receiving garbage packets from the network. The packets with correct node address or multicast packets may still be undesirable, if the host is being bombarded with them due to a network malfunction. In the interface, the on-board processor is not involved in transferring packets into the interface memory, and can receive a number of packets without being interrupted by the NAC. In the prototype NAB using a fast microprocessor such as AMD 29000, the RAR matching algorithm runs fast enough to keep up with a packet rate in excess of 185,000 packets/sec, which is more than the maximum packet rate of 160,000 packets/sec of a 100-MB FDDI network assuming all packets are minimum size VMTP packets. As long as the network throughput does not exceed adapter's processing capacity, the buffer memory cannot get filled up with packets that do not match an RAR. Thus, the availability of buffer memory to receive authorized packets remains independent of the network pollution that may be taking place.

[3] Calculated by assuming packet size of 1024 bytes and block transfer rate of 320 megabits, available with the serial mode transfer on the VME bus.

4.3.2 The Packet Processing Pipeline

The network adapter incorporates several hardware modules that constitute an interlocked *packet pipeline*, performing checksumming, encryption and network access on transmission, and the reverse on reception, all operating at the network data rate. Each stage in this pipeline is a simple operation, such as memory read/write, compare, and one's complement, as needed for these algorithms. The data transfer unit between stages is 32 bits. A word is prefetched at each stage and stored until the function is completed. A short FIFO is included in the NAC to allow for setup delays for decryption and node address recognition on packet reception. The on-board processor updates packet headers and queues the packets for transmission as well as processes packets on reception.

The common format used for all VMTP packet types, shown in Figure 4.3, is ordered so as to minimize processing delay, overhead and complexity. The

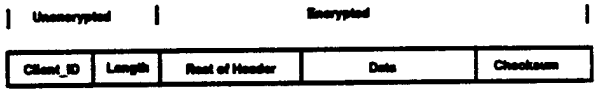

Figure 4.3: A VMTP packet

first field specifies the client entity associated with the message transaction, the indicator for the decryption key to be used with the packet. The second field indicates the length of the packet and various address qualifiers. It is sent in the clear, providing the network adapter with some time to access the decryption key before it encounters the encrypted portion of the packet. Various other fields follow that indicate the VMTP type of packet, its packet group and its position within the packet group. The order of these fields also corresponds to the order in which the reception logic needs to map the packet to a particular client, packet group and data within the packet group. The checksum field is located at the end of the packet, (rather than in the header) so that it can be calculated "on the fly" on both transmission and reception. The length field specifies the offset of the

checksum field from the beginning of the packet.

Reception processing proceeds as follows. When a packet starts to arrive at the network access controller, it checks the destination address and then forwards the first words of the packet to the first stage of the decryption hardware. At this stage, the first data word received is delayed until a decryption key is located in the decryption key cache (indexed using a hardware hashing function on the destination address). If the key is located, the data proceeds through the decryption stages, each one of them performing a function in the decryption algorithm.[4] If the key is not located or decryption is not needed, the data stream proceeds to the checksumming stage with the decryption stages disabled. The start and end points for checksumming of data bytes is specified by the offset values initialized by the user. Each data word is passed through a 1's complement sum to form two 16-bit partial checksums.[5] After receiving a packet, a 32-bit checksum is calculated from the partial checksum and added to the buffer. The final step is to write the data word in the serial staging register. When the end of the packet is received, the data is moved to a receive buffer address provided by the serial-port controller. The empty buffer addresses are provided to the controller by the on-board processor. The received buffer is queued for header processing by the on-board general-purpose processor.

The received packet is lost if the buffer memory is full. A packet is also lost whenever the transmission is interrupted due to the lack of data in any stage of the pipeline. Both these unusual conditions require flushing and restarting of the pipeline, a task that is handled by the processor.

A packet pipeline as described increases the performance due to the following reasons. The processing overhead of performing functions such as checksumming and encryption is hidden in the copying operation, i.e., the copying of packets from the network to the buffer memory. Otherwise, additional memory references would be required to perform checksumming or encryption, wasting bus capacity as well as memory bandwidth, which are critical resources in a multiprocessor system (and in a network adapter). The latency due to the packet pipeline is kept low by using very few stages, and by restricting the width of the pipe. In the prototype, the pipeline uses fewer than 5 stages and a 32-bit wide data path. In contrast, if a larger unit of inter-stage data transfer, such as a full packet, were used, the time to fill up the pipeline would be larger, thus increasing the latency to transmit a packet. Finally, many simple stages, implemented in hardware, and running in parallel provide better performance than a single fast processor performing the same functions.

The fine-grained, tightly synchronized processing achieved by the packet processing pipeline suggests that comparable performance would be difficult, if not impossible, to achieve with multiple general-purpose processors working in parallel. In particular, with a single packet transmission or reception, the multiprocessor solution would perform at the level of a single processor, which is far below the performance of the NAB pipeline. With multi-packet transmissions and receptions, each processor has to wait for the reception of "its" packet before it can start processing the packet. The combination of waiting for these large units of data, the processing overhead per packet and the limited number of packets in a "blast" makes this approach unattractive.

A general-purpose multiprocessor solution is further hampered by the degree of correlation observed between successive packets received from the network. For example, the measurements of the VMTP traffic on a 10-Mb Ethernet [11] show that statistically the packet currently being received belongs to the same message transaction to which the previous packet belongs. Because packets in the same message transaction share control block information at sender and receiver, this correlation results in interference among multiple processors for both memory access and synchronization. Thus, a general multiprocessor architecture in which each processor works on a different packet appears significantly inferior to the pipelined architecture of the NAB.

We have placed the packet pipeline between the buffer memory and the network, forcing it to run at the network data rate. This alternative would be to place it between the host bus and the buffer memory. The alternative placement is undesirable on many counts. Burst transfer rates on host buses are much higher than network rates, forcing one to operate the pipeline at higher data rates. The synchronization of the pipeline operation with the bus is also more difficult because the interface typically involves transfer of data from memory as well as con-

[4] For the prototype, we have used a single cycle of product cipher algorithm that is similar to the DES standard.

[5] The complete checksum algorithm used is described in the specifications for VMTP [9].

trol information from one or more host processors. Moreover, on reception we do not know where to put data until the received packet is decrypted, further complicating the pipeline design. Last but not least, placing the pipeline at the host-end precludes strict implementation of a network firewall; data is transferred across the system bus before detecting checksum errors or that the packet is not wanted.

Most transport-level protocols preclude strict pipeline processing as described here because the checksum field is included in the header. As a consequence, the checksum has to be calculated before the packet header can be transmitted, preventing significant pipelining. VMTP appends the checksum to the end of the packet to allow pipelining. It also uses a 32-bit checksum for compatibility with the pipeline width, as compared to the conventional 16-bit checksums used in TCP.

The main disadvantage of the packet pipeline described here is that it is protocol-specific. The other disadvantage is that the pipeline has to operate at the network data rate, although the functions implemented in the pipeline are simple and one could use a CMOS logic up to 1 gigabit/sec rate. Note that each stage of the pipeline processes one word (32-bit wide) at a time, and hence is clocked at a rate $10^9/32$, i.e. 32-Mhz, well within the range of current CMOS technology.

4.4 NAB: Software

4.4.1 The On-board General-Purpose Processor

The network adapter includes a general-purpose processor that manages the buffer memory, the host block copier and the packet processing pipeline. A general-purpose processor was chosen for these functions because they are significantly more complex than other functions, offer less performance benefits in a specialized hardware realization, and were less understood than the packet pipeline and block copier functions. The cost of processing performed by the NAB processor seems to be equally shared among various functions rather than being dominated by a few performance critical functions, thus reducing any potential benefit of the hardware realization. Moreover, using a general-purpose processor instead of a special-purpose protocol processor allows one to benefit from the continual advances in microprocessor technology.

With the general-purpose processor, we can program the basic queueing and dequeueing required to manage the buffer memory as well as any functions required by the protocol. However, we have taken care to partition the processing between this on-board processor and the host processors as follows. The on-board processor executes the "common case" packet processing, namely, the error-free transmission of packets; whereas, a host processor executes the "rare" case packet processing, which includes functions such as acknowledgement and retransmission of messages. The distinction is made primarily to simplify and thus execute faster the functions critical for improving the whole transmission or reception. For example, when transmitting a VMTP packet group, the processor updates the header delivery mask and appends a new buffer of data to the packet header between packets, queuing the new packet for the packet pipeline. However, one of the host processors is charged with forming the header for the first packet of the group because this task is more complex and requires close interactions with the user process and the operating system.

The on-board processor keeps a queue of TARs and RARs waiting for processing. The TARs for sending a single packet are directly queued for transmission. The others are added to a circular queue of TARs scanned periodically by a scheduler. The scheduling of TARs take into account the priority and the inter-packet timing gap constraints attached to each TAR. Typically, each ready TAR gets to send a packet before the next one can be sent from the same group of packets.

A hashing algorithm is used to match the incoming packet's destination address to the locally acceptable addresses. The hash table entry contains the acceptable destination, local host number, local process identifier, and a pointer to a queue of waiting RARs. For the first packet in a group of packets, the packet's destination address is first used to locate the appropriate queue of RARs and then to match the appropriate RAR. The matched RAR is assigned to this group of packets and added to a list of recently matched RARs. This list is scanned first for the subsequent packets of this group of packets. In addition to filtering done by matching of RARs, it is also possible to filter packets based on a small number of (network-level or transport-level) source addresses from which all traffic is refused. This facility imple-

ments the "firewall" function which allows the NAB resources to be protected to some extent from failures or malicious behavior of a remote host processor or a user process. The selection of the appropriate source address to block on is done by a local host processor monitoring the traffic from the network.

The RARs and TARs remain indefinitely at the NAB until either the operation requested is complete or until a host requests them back by issuing a reset of the NAB. This "soft" reset operation allows a host to clear the queue of waiting RARs or TARs bound to a specific destination, or all waiting RARs and TARs at the NAB. Explicit timeouts per TAR or per RAR, based on the timeout values provided by a host, is an alternative way of handling the problem of unused RARs or TARs. These explicit requests to flush out the unused records minimize the timer-related NAB processing, which is a non-trivial overhead especially at servers that may have many outstanding RARs and TARs. We expect the frequency of such flushes to be small, justifying the additional overhead of the "soft" resets.

4.4.2 Host-Adapter Interface: Details

In this section, we focus on the message formats and the semantics that define host-device interaction. Figure 4.4 shows the details of a format of messages exchanged between device and host.

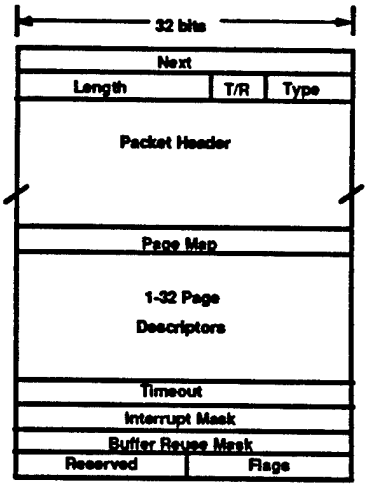

Figure 4.4: An UNDIP Message

Type	Function
000	TAR (RAR) request with buffers
001	TAR (RAR) response with buffers
010	TAR (RAR) request with data
011	TAR (RAR) response with data
100	TAR request with multicast response
101	CR request and response
110	TAR and RAR request with buffers
111	TAR and RAR request with data

Figure 4.5: Types of Messages in UNDIP

Two basic formats used are Transmit Authorization Record (TAR) and Receive Authorization Record (RAR), distinguished by the T/R bit. TAR is used to control transmission from a host to network and RAR is used to control reception from a network to host. The Packet Header field in a TAR contains the header of the first packet to be sent or received. The Buffer List contains a list of pointers to host memory pages, which in turn contain data to send or received data. The Next link facilitates storing of records in a queue. Messages recognized in UNDIP are distinguished by the Type field (Figure 4.5).

Basic Procedure

A typical procedure used for data transmission and reception is as follows. To transmit data, the host sends to the device a TAR (Type=000) with the packet header and a list of descriptors of host memory pages containing data. The packet header is used by device as the first packet header. It is also used with minor changes as the packet header for subsequent packets formed out of the same data segment. A host can line up more than one TAR at a time and the device multiplexes their transmissions. The completion of transmission is signalled by the device returning the TAR to the originating host.

To receive data, the host sends to the device an RAR (Type=000) with the packet header of the first packet expected to be received and a list of pointers to memory pages reserved for receiving data. The packet header contains a specific or partially-defined source node address and transport-level identifiers. For each incoming packet, the device matches the received packet to an RAR and, on a successful match, writes the received data to host memory pages pointed to by the matched RAR. If the buffer

list is exhausted or transaction is complete, the RAR is returned to its originating host processor. Each transport-level identifier may have multiple RARs waiting for packets to arrive. In this basic procedure, no matter how large the data segment is, only four host-device messages are exchanged.

Enhancements

The basic procedure is augmented with the following features that further minimize host-device interaction. The interface allows direct inclusion of small amounts of data in a TAR and an RAR (Type=010 and 011, respectively). Transaction style of interactions, typical for many distributed applications, are supported by a message type that integrates the functions of TAR and RAR. A client issues a TAR/RAR (Type=111) to send as well as to receive data. The host memory pages are reused after data transmission as buffers for receiving data. After data transmission, this message is transformed and saved by the device to act as an RAR waiting for data. Additional interesting features supported are multicasting, timeouts to return unused TARs and RARs, flexible masking of packet interrupts, and buffer management policies designed in part for large transfers of data and real-time traffic such as packetized video. These features are described in detail in [40].

Error Detection and Retransmissions

The interface allows a choice in locating error-control and retransmission functions among device and host. For the environment where transmission errors are frequent, such as a packet radio network, these functions could be located on the adapter. This would eliminate data retransmission over the host bus and memory but would also increase packet processing load on the adapter.

The location is specified with an EC bit in the Flag field of TAR or RAR. A TAR with the EC bit set specifies that device should retain buffer descriptors until acknowledgment arrives and should handle retransmissions. An RAR with the EC bit set specifies that device should generate acknowledgment packets. The details of acknowledgment and retransmission procedures depend on the transport protocol used.

Flow Control

Rate-based flow control, used in recent transport protocols, is difficult to implement in host software. The typical inter-packet gap desired is too small for a task switch and inserting idle software loop to generate the gap is not only inefficient but is unable to generate precise gaps. The device has the knowledge of demands of processors (processes) and the network load and can implement flow-control in hardware, making it ideal for performing flow control. A host sets the flow control policy and adjusts inter-packet gap requirements, which are conveyed to device, using UNDIP, in each TAR or RAR. The device ensures gaps in packet transmission by either waiting for the duration, multiplexing transmissions from two local sources, or by giving up its access rights to link and then waiting for the next chance.

The flow of TARs and RARs between host and device is self-regulated. If the device runs out of memory space for storing TARs and RARs, it returns them to the host, marking them as unsuccessful. This simple model of flow restriction is sufficient to build allocation policies in device software that prevent unfairness to some applications.

4.5 Summary

In this chapter, we have described NAB architecture for host interfaces. The key performance increasing features of NAB architecture are

Video-RAM based memory organization The memory organization achieves true concurrency in processing packets with their transfer through the adapter memory.

Packet Processing Pipeline Packet processing pipeline reduces the total precessing time by folding the cost of checksum and encryption functions with the transfer of data. In order to facillitate the pipelining, transport-level packets should have end-to-end checksum following the data portion.

Header prediction algorithm Header prediction algorithm reduces processing time because of the observed locality in network traffic. In conventional interfaces, the improvement is not significant because header processing cost is a small part of the total processing time. In NAB,

the performance improves significantly because NAB has already reduced other packet processing costs leaving header processing as a potential bottleneck.

Host-to-Adapter interface The host-to-adapter interface treats small and large data transfers differently, handling small transfers with a programmed I/O interface and large transfers with "intelligent" DMA interface. The programmed I/O interface minimizes delay for small packets. The DMA interface for large transfers minimizes cache and system bus traffic. Transport protocol facilitates easy recognition of small packets requiring low latency.

The adapter-to-host interface interrupts host on a data segment boundary and not for each packet transferred. For multi-packet transfers, this significantly reduces interrupt processing overhead, which is especially costly in multiprocessor systems, as one may need to trap data and code in the cache to handle interrupts.

The authorization model for reception allows a host to continue functioning even when being bombarded by packets from one or more hosts. This "firewall" function is essential for connecting hosts to a high-speed network.

Chapter 5

Performance Analysis

5.1 Introduction

The two important cases to consider are throughput for large amounts data transfer and response time for a small amount of data transfer. The throughput is measured as number of user-level data bytes transferred divided by the total time elapsed between user call to kernel and kernel's response indicating successful transfer of data and acknowledgment from the remote end. The response time for small transfers is the total time elapsed between a user's kernel call and its response. Additional measure of interest is the processing load on server imposed by communication protocols. With lower processing load per connection, a server will accommodate more connections without being a bottleneck.

The performance is based on the timing information obtained from the prototype NAB design, and software processing time estimates are based on the VMTP implementation in the V distributed system. The prototype uses Motorola's M68020 processor, which is rated as a 2-MIPS processor. The performance of prototype using a more powerful processor is studied by running the NAB software on a chip-level simulator for AMD's Streamlined Instruction Processor (AMD29000), which is rated as 17-MIPS processor. A 100 megabits/sec FDDI ring is assumed for estimating the network delay.

5.2 Delay for Short Packets

Delay is the key issue in considering the performance for short packets. Figure 5.1 shows the estimated request-response time for a VMTP message transaction with no segment data, which results in a packet of minimum length, i.e., 64 bytes plus a network header.

The total delay includes the network latency, which is the waiting time to access the network, the packet transmission time, the bus transfer time, and the processing times at sender and receiver for both a request followed by a response packet. We assume the network latency of 100 microseconds, reflecting the average round-trip time in a lightly loaded FDDI ring of moderate size. The packet size is 64 bytes of VMTP packet and network header is less than 16 bytes. The network delay, ignoring the small propagation delay, is the network latency plus the packet transmission time, 100 + (80 * 8 /100) microseconds, which is 106.4 microseconds. The estimated NAB processing time at the sender is 50 microseconds, based on approximately 150 instructions needed to process the TAR, and to schedule the transmission. The estimated NAB processing time at the receiver, which includes the RAR matching time and the processing of a received packet, is 50 microseconds, assuming that an RAR is found. The packet processing times at a receiver and a sender, estimated from the actual CPU time observed for Send-Receive-Reply IPC in the V operating system, are approximately 275 and 442 microseconds, respectively. These measurements are for a request followed by a response packet, both without a data segment between two SUN 3 workstations communicating over the 10-MB Ethernet. We measured these values with the help of fine-grained timer (10 μs interval) available from Peter Danzig [20]. Using these values, we find that the total request-response time is approximately 1.14 milliseconds.

In Figure 5.1, we note that only about 12% of the total request-response time is spent in network adapters. The dominant factor is the host processing of the request and the response packet (about 88%). The network latency time and the NAB processing

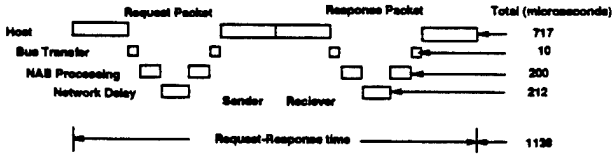

Figure 5.1: Response time for single short packet transfer

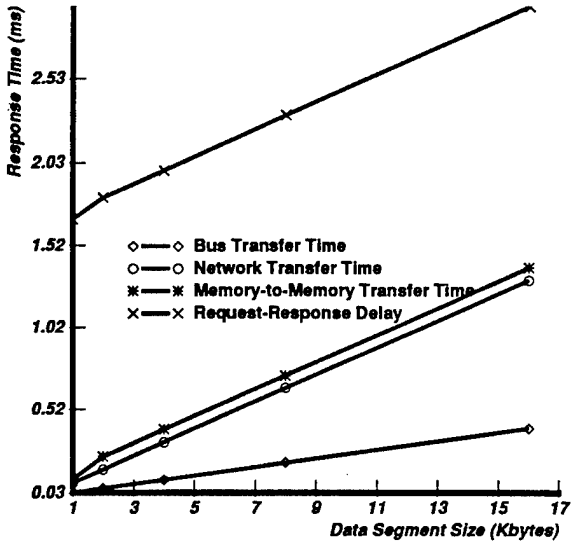

Figure 5.2: Response time for large data transfers

are each about 5.6% of the total delay. Thus, further optimization of the NAB architecture for this case would yield marginal returns.

5.3 Throughput for Packet Groups

Figure 5.2 shows the total request-response time for transferring large amounts of data. To estimate this delay, we first consider the delay for host memory-to-memory data transfer. The request-response time is obtained by adding to this delay the host processing times at the sender and the receiver, plus the start-up time at the NAB for beginning a packet group transmission. Memory-to-memory transfer time is the network transmission time for a packet group, with the assumption that the packet processing required is less than one packet transmission time on the network. This is indeed the case in our design, once a packet group transmission begins. We neglect the small propagation delay of the medium, and we neglect the network latency, which in this case is small compared to the total data transmission time and could be made smaller by sending multiple packets in each access to the network. The packet size is 1024 bytes and the bus transfer rate is 320 megabit/sec. Thus, the buffering latency for the first packet is 25.6 microseconds. Finally, we conservatively estimate 50 microseconds as the start-up time at the NAB for beginning the packet group transmission.

From Figure 5.2, we calculate that the (expected) effective throughput for 16 kilobyte transfers is 44.3 megabit/sec. The memory-to-memory transfer rate is 94.5 megabit/sec. The buffering latency of the first packet and the start-up time are, respectively, 0.8% and 1.7% of the total request-response time for reading 16 kilobytes. From Figures 5.1 and 5.2, we see that in the NAB architecture the Request-Response delay for both short and large data transfers, is dominated by the host processing time.

In Figure 5.3, we show the impact that a NAB interface will have on how much server capacity per connection is utilized. The reduction in CPU cycles effected by NAB is shown as a percentage of CPU cycles currently required. The 16-KByte transfer of data results in using only 10% of the CPU cycles currently measured as being devoted to that operation. The current cycles used are obtained on a Sun 3/75 workstation with a program that measures CPU cycles while repeatedly reading 16 KBytes. The cycles needed with the NAB are measured on an VMP processor equipped with hardware counters, counting cache and other memory references.

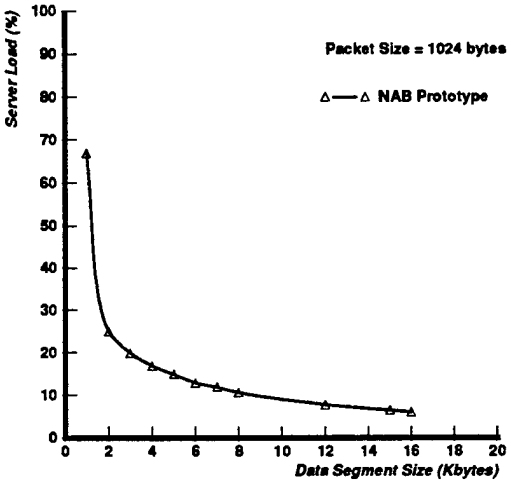

Figure 5.3: Server load with NAB relative to what is imposed currently on a Sun 3/75 workstation.

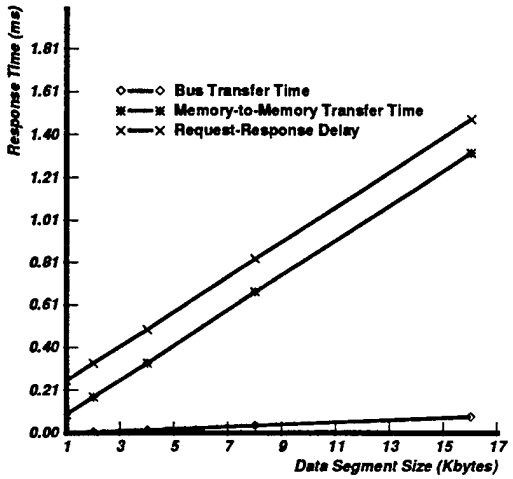

Figure 5.4: Response time for NAB using a 20-MIPS processor.

5.4 Effect of Improving Processor Technology

The performance would improve if a powerful CPU is used both as the on-board processor and as a host processor. The faster CPU reduces the processing times at hosts and the start-up time for beginning the packet group transmission. Moreover, the reduced packet processing time at NAB, the cost of which is hidden in our design in the packet transmission time, allows one to use shorter packets with shorter transmission times; the shorter packets reduce the buffer latency time further decreasing the overall delay. The future workstations are also expected to use newer and faster system bus designs such as the FutureBus [3], which can transfer data at 1.6 gigabit/sec. Taking all these factors in account we estimated Request-Response delay for large data transfers using 20-MIPS processors in Figure 5.4. The packet size is 1024 bytes and the network speed is 100 megabit/sec. The processing times for processors with greater than 2 MIPS power are derived from our measurements on an MC68020, 2-MIPS machine, by reducing these figures by a factor proportional to the processor power relative to an MC68020.

In Figure 5.4, we note that the estimated response time for 16 kilobyte transfers with a 20-MIPS CPU and a faster bus is 1.48 ms. This means an effective throughput of 88.6 megabit/sec. The memory-to-memory transfer time rate is within 1% of the transmission time over a network of capacity 100 megabits/sec.

A well-balanced NAB architecture would match the NAB processor's capacity to the network data transfer rate, such that the time for processing a packet equals the time of receiving or transmitting a packet. In Figure 5.5, we show the expected throughput as a function of processing power. We assume that that the system remains well-balanced, i.e., packet's transmission time equals its processing time at the adapter. The effective throughput is calculated for transfer of 16 kilobytes using maximum packet size of 1 kilobyte. The request-response throughput includes host processing time, which is 1575 microseconds at 2 MIPS, and which is proportionally reduced for a faster processor. The packet processing time within a packet group is estimated to take 75 microseconds for a 2-MIPS processor, based on approximately 150 instructions required for on board processing at the receiver. The processing time is reduced proportionally to obtain processing times (and thus transmission time) as the processing power increases. At 20-MIPS, as calculated from the the memory-to-memory transfer rate is approximately 1 gigabits/sec. The actual system performance is likely to be limited by the system bus capacity or the mem-

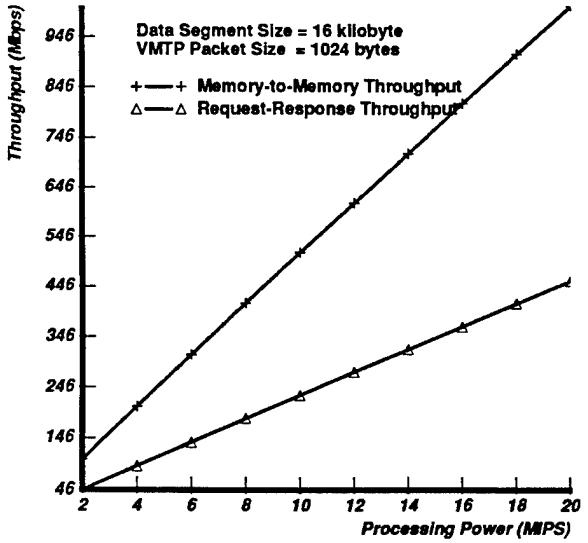

Figure 5.5: Estimated throughput for NAB as a function of adapter processor's power in MIPS.

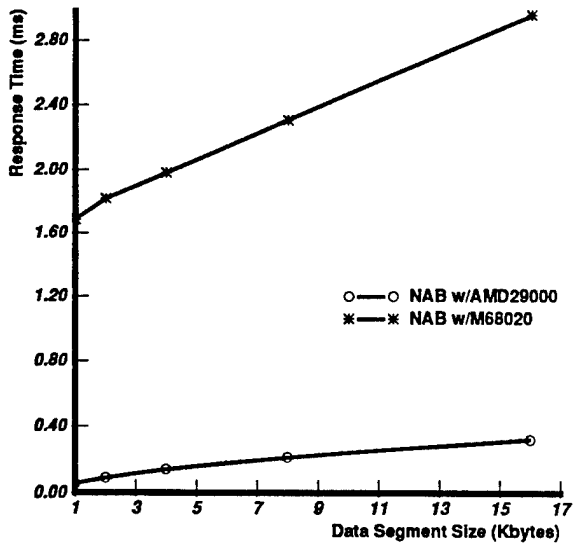

Figure 5.6: Response times for the AMD 29000 NAB.

ory bandwidth; both of these being assumed here to be adequate.

We used a chip-level simulator of AMD's 32 bit RISC microprocessor in order to further investigate how the performance scales with an increase in CPU power. To run the NAB software on the simulator, we had to change only one low-level routine of the NAB's simple operating kernel, which is nothing but a collection of low-level interrupt handlers and a prioritized event scheduler. The observed throughput is shown in Figure 5.6. The response time measured excludes kernel overhead. The network connecting these simulated adapters is assumed to be a gigabit/sec link. The response time obtained is compared with that observed with that of the prototype using M68020, and also with that obtained with current host interfaces on a Sun 3/75 machine running VMTP software on the V system.

5.5 Cost and Performance Analysis

In this section we study the performance benefits and cost of key aspects of NAB, namely, packet processing pipeline, memory organization, and on-board processor.

5.5.1 Packet Processing Pipeline

We observed the performance degradation that occurs when on-board processor is forced to perform each of the function performed otherwise in the pipeline. In Figure 5.7, we show estimated response time for three different configurations with NAB Software running on the AMD processor simulator. Three configurations differ by the functions added to the NAB software. The first one performs neither checksumming nor encryption of data. The second one performs checksumming in software. The third one performs checksumming as well as encryption in software. As expected, response time is highest for the third configuration. The response time for 16 KBytes of data transfer in the second and the third configuration are, respectively, 4.4 times and 9.2 times that in the first one. Because checksumming and encryption in software requires accessing each data word, it is not surprising to find that response times increases far more rapidly for the organizations that perform these functions in software.

In the prototype, the cost of the components for the pipeline is about $150. This is about the 1/8 the total cost of components with MC68020 as the onboard processor, and 1/15 the total cost of components, if AMD29000 is used as the onboard processor. The cost of other components are roughly

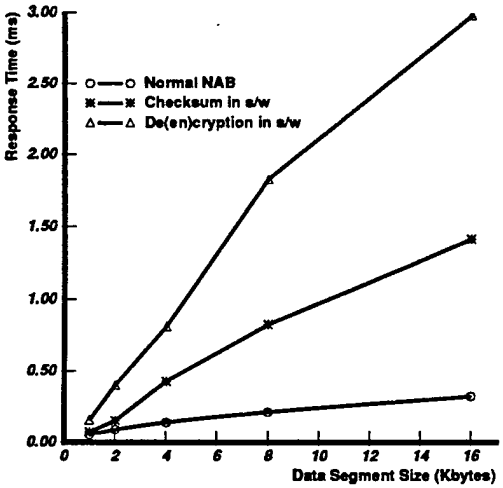

Figure 5.7: Response times when checksumming and encryption is not performed in the packet processing pipeline.

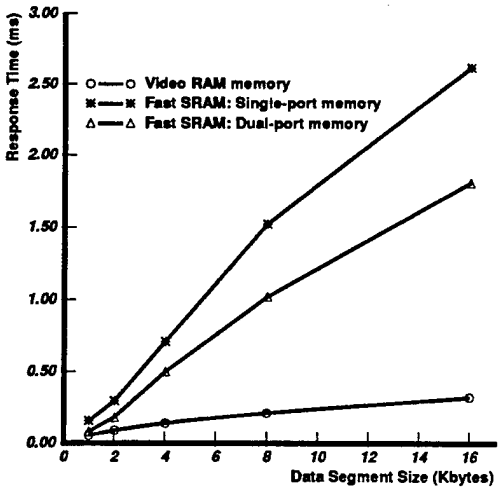

Figure 5.8: Response times of NAB with various memory organizations.

$500 for memory-related components, and $400 for AMD29000 processor. The cost for FDDI network controller and transciever circuits is roughly $700 and the remaining cost is for other supporting circuits such as host bus interface and timer chip. To the component cost, one should add the cost of PCB and wiring of the logic required for building a pipeline. In the prototype, the logic circuits for pipeline takes up about 2/5 of the overall board space.

In summary, with an additional cost that ranges from 1/8-2/5 of the total cost one could obtain 10 times better performance by building the pipeline.

5.5.2 Memory Organization

We examined the response time for three different memory organizations in order to evaluate the effectiveness of using VRAM-based memory. The first one is the normal NAB memory that uses VRAM components. The other two uses fast SRAM components with cycle time of 80 ns. One of these is a single-port organization where access from processor, host bus and network are arbitrated. The other one used dual-port organization where access from host and processor are made to one port and the network are made to the other port. The performance differs in these cases only because of the bus interference and arbitration overhead due to simultaneous attempts to access the memory. Figure 5.8 shows the response times for these three memory configurations as the size of data segment transferred increases. Response times were measured by running the NAB software on the AMD29000 simulator with the appropriate memory model. The reported measurements are averages observed of 10 experiments, each of which performed 1000 data transfer operations without packet loss. As expected, the VRAM-based memory has the lowest response time.

When we consider the cost of components, VRAM-based memory is even more attractive. Fast SRAM components of a comparable size and speed are about 3 times more expensive than the VRAMs used in the prototype. The board space and wiring costs for all three are almost the same. Moreover, the memory is roughly about 3/5 of the total cost of the board. A minor disadvantage is the increased power consumption when VRAM-based memory is used, which reflects in the increased cost of the power supply. A VRAM IC used in the prototype consumes about 1.5 times the power consumption of the fast SRAM component considered here.

5.5.3 Onboard Intelligence

We quantify here performance benefits of having onboard intelligence. We consider three different levels of onboard intelligence. Note that the level numbers are not to be confused with the layer of protocol it implements. Intelligence Level 0 is the one that uses a dumb adapter. An example is a sun 3/50 machine with an Ethernet controller chip. In this system all VMTP functions are implemented in host software. Intelligence Level 1 is the one with the cut-through adapter, described in Section 3.3.2. In this option, the adapter does only checksumming and encryption and has a host that performs the remaining functions of VMTP. We assume the best case for this adapter. The best case is that packet processing software used at host processes packets as efficiently as the NAB software does; The only difference is the added cost of process switching and the interference in processor-memory traffic due to packet transfers over the host bus. Intelligence Level 2 is the intelligent adapter performing checksumming, encryption and packetization for intermediate packets. We further consider three variants of adapter architecture for the Level 2 adapter. In the first variant, the internal architecture used is the NAB architecture. In the second variant, NAB minus the block transfer protocol of the host bus is used. The last variant is the intelligent adapter with the conventional memory and bus organization. All three variants run the NAB software with minor changes.

The performance of the first system, Level 0 adapter, is measured on the V kernel running on a Sun 3/50. The performance of other systems are estimated with the the NAB software running on the simulator of NAB using AMD29000 processor. The performance of Level 2 adapter with a conventional internal architecture is measured by changing the memory model used for the simulator. A comparison of the last three systems show how much performance improvement is the result of using NAB as an internal architecture.

Figure 5.9 shows the performance and hardware cost of these five systems. The first column of the figure shows the expected response time of these systems for a 16-KByte write operation. The second column shows the expected cost of the adapter used in these system. The cost estimates are based on the cost of components and the board space needed. The costs are normalized by the cost of an Ethernet controller chip and the circuitry needed to interface with the chip.

The type of onboard intelligence selected should depend on the class of host machine used. Host machines on a network can be categorized in three classes: network terminals, workstations, and powerful servers. The choice is clear-cut for low cost network terminals as well as powerful server machines. In a network terminal, the relative cost of adapter is significant and since the performance is not limited by the adapter, a dumb adapter (Level 0) is the appropriate choice. For a server machine, such as a Cray machine or Alliant series multiprocessors, the additional cost of general-purpose processor on the adapter is a fraction of the total system cost. For servers, there are multiple clients waiting to get service. Because of this, minimizing the wastage of servers' critical resources should be a high priority. Thus, the relative cost and high performance desired both argue in favor of a NAB-like adapter.

In workstations, the level of onboard intelligence to be used depends on a number of factors. Proponents of dumb (and simple) interfaces argue that because a workstation is a single user environment, the host processor is frequently available to perform communication tasks without significantly lowering the overall performance. Moreover, it is argued that a workstation is generally cheap enough to make network adapter cost significant. For example, currently a low-cost workstation from Sun costs $8,000 whereas the FDDI interface costs $3,000-$5,000. These arguments seem to favor using a dumb adapter (Level 0).

But, the future trend in workstation technology favors using intelligent adapters even for workstations. Microprocessor technology is pushing towards lower instruction cycle times, making host bus and memory critical resources. Moreover, the demand for desk-top computing power and particularly high quality display continues to increase the price of a workstation. In contrast, the price of adapter with an onboard general processor remains more static, since the onboard processor has to be only powerful enough to keep up with the network's data rate. And, upgrading network technology to higher rates proceeds at a slower pace, because it entails upgrading all the connecting host interfaces at a time. We found that a NAB built with M68020 processor would be adequate (Figure 5.6) for a 100 Mbps link. Because the adapter processor need not be changed or redesigned unless network is, one can also take advantage of rapid decline in prices of old processors that occurs when faster

Onboard Intelligence of Adapter	Response Time for 16-KByte Writes (ms)	Hardware Cost of Adapter Used
Level 0	121.5	1
Level 1	8.1	3
Level 2 (Var. I)	2.9	14
Level 2 (Var. II)	5.5	12.3
Level 2 (Var. III)	110	8.5

Figure 5.9: Response times and relative costs of different adapter architectures.

processors are offered. The list price of a DECstation 3100 with color monitor (rated at 10 MIPS) is in the $50,000 range. For this workstation, the additional dollars (est. $500) required to build a NAB like interface as opposed to the cut-through adapter is about one percent of the total cost. Thus, all of the future trends in workstation technology seems to favor onboard intelligence. But, for low-cost workstations with low performance (2-5 MIPS region), an adapter peforming only checksumming and encryption functions (Level 1) could be a good choice.

5.6 A Summary of Results

The expected performance of NAB prototype shows a large increase in performance compared to current host interfaces. For instance, the effective throughput of MC68000 NAB for a transfer of 16 KBytes is 45 Mbps at the user-level. Multiple connections through the NAB would have the aggregate throughput of no less than 90 Mbps. The expected throughput for AMD29000 NAB is 200 Mbps at user level and 450 Mbps aggregate throughput. The throughput increased tenfold to hundredfold as compared to the best throughput currently available on hosts implementing VMTP.

The expected response time for small transfer also shows improvement. The request-response time for small transfers would be 1.14 ms for MC68000 NAB.

One of the stated goals for having onboard intelligence was to save the host bus cycles used for communications. The goal is indeed met. As shown in Figure 5.3, the NAB brings the host bus capacity used for communication to 10%-70% of that used with a dumb adapter.

The performance analysis of having onboard intelligence, shown in Figure 5.9, indicates that the NAB adapter has the lowest response time compared to a dumb adapter, an intelligent adapter with a conventional architecture, and a cut-through adapter. The cost of hardware, which is typically a small part of the total cost of a host, is the highest for the NAB adapter. The NAB architecture seems ideal for an adapter for a server machine because in this environment the increase in performance and the saving in host bus cycles typically outweigh the cost consideration.

We also quantified performance improvements obtained with the pipelined processing (Section 5.5.1) and with the Video RAM based memory organization (Section 5.5.2).

Chapter 6

Related Works

6.1 Introduction

The interfacing of hosts to networks is an important research issue that was not appreciated in the early days of networking. At that time, the focus was on studying the problems of sharing communication links and switches. There were two major reasons for this lack of interest. The communication links were of relatively low speed and the major network-wide applications of that age, namely, electronic mail and remote-login, did not demand high throughput. Two subsequent developments have contributed the most to the current focus on the host interfaces.

The development of distributed systems demanded high performance at user-level. Distributed operating systems made geographical distance and machine identities transparent to users. These operating systems, designed from scratch, made communication services the most fundamental aspect of their structures. With this development, the need to improve performance became noticeable. Some notable example of distributed operating systems are DEMOS [2], Thoth [5], Accent [52], V [8], and Locus [50] operating systems.

Multi-megabit local area networks such as Ethernet and developments in optical fiber technology made raw bandwidth cheap and plentiful. With this came the realization that the performance available to users was quite below what a network offers and what a user wants. This fact was observed in a number of studies of network traffic and distributed systems [43, 41, 55].

6.2 Better Software

Past efforts made for improving user-level performance may be categorized by the relative emphasis placed on either the software or the hardware changes. The ones that relied solely on software changes, leaving the underlying hardware unchanged, are discussed first. The others are discussed in the following section.

One of the recent, successful effort that relied only on software improvements is by Van Jacobson and Mike Karel [35, 36, 37]. They made three major changes in the TCP/IP implementation on BSD 4.2 Unix, improving its performance. The first and perhaps the most important change made was to remove redundant, hidden copying of network-bound data in the operating system. The second change they made was to improve the interface between the kernel and the device driver to handle data passed between them in larger chunks. The third innovative change made was to change the header processing algorithm. It made use of a hint produced by the examination of the last received packet. With the changes, throughput for transferring 32 Kbytes of data over a 10-MB Ethernet increased from 2 Mbits/s to nearly 8.4 Mbits/s.

Another interesting effort solely relying on software changes was reported by Carter and Zwanaepoel [4]. It managed to increase the VMTP throughput from 4.5 Mbits/s to nearly 8.5 Mbits/s by making use of locality in network traffic to eliminate data movement of large packets. The basic observation made by them was that the data contained in the received packet frequently goes to the next location in the host memory. So by default, the data received was transferred directly to the memory location predicted by the examination of the last packet received. Corrective actions need to be taken if the packet did not belong there, a fact that is discovered on the processing of its header. The procedure avoids the additional data move but does so at a considerable cost required for

correcting prediction failures.

6.3 Hardware Support for Protocols

A first attempt at building a full transport protocol in a VLSI was made at Mitre Corporation [46]. They developed a single chip implementing TCP/IP standard. There were two major problems with this chip. It was slow and it handled at most two connections. Both these problems were caused by its internal architecture. The architecture used a standard microprocessor CPU unit and on-chip microcode store that contained TCP/IP program.

The second interesting effort is Greg Chesson's Protocol Engine project [13], currently in the development phase. His approach is based on a lightweight transport protocol called XTP [14], developed specifically to make its hardware implementation easy. Currently, they have an architectural design of the PE ready and a software implementation/specification for XTP. The reported details about the design indicates that the PE is a specialized processor to be implemented in four VLSI chips. One unusual feature of the design is that it plans to use an additional program address register to effect fast context switching between software modules implementing transmit and receive procedures. Another special feature planned is the hardware to assist in searching a connection control block from the destination address of a packet. Other transport-level functions such as packet retransmissions, error control, and flow control are planned to be performed with a microcode.

Researchers from NTT Laboratories, Tokyo, have reported the design of a VLSI chip for X.25 layer 2 and 3 protocols [32]. The VLSI employs parallel processing between layers 2 and 3 of both the receive and the transmit side. It also provides complete hardware-handling of data transfer control and buffer management functions, and directly executes high-level protocol description language for flexible protocol implementations. The performance of the VLSI chip was estimated to handle packets at data rates up to 50 Mbits/s. The prototype built with CMOS technology has about 140K transistor-logic circuits, about 168 Kbits of ROM and 27 Kbits of RAM integrated on the chip. The VLSI chip runs at maximum clock rate of 32 MHz and is housed in 208 pin grid array package.

Krishnakumar, Krishnamurthy and Sabnani from AT&T Bell Laboratories [42] have proposed a different approach in building a protocol VLSI. They proposed modeling the core of a protocol formally with a collection of Finite State Machines (FSMs) [54] communicating with one another through synchronous message exchange. This model captures coordination between peer entities. Low-level details such as message formats and timer operations are also specified formally. The overall architecture consists of a major unit that performs core functions modeled as communicating FSMs, and a number of satellite units such as the message parser, message assembler, and the memory controller. The VLSI design is generated automatically from a formal specification of a protocol. Currently, a prototype is being built, that is expected to show at least as much throughput as obtained with the X.25 VLSI from NTT.

With the exception of A. Spector's PhD Thesis [57], no efforts have been made for hardware designed specifically to reduce latency for exchanging small amounts of data. A. Spector modified an ALTO machine, developed at XEROX's Palo Alto Research Center in the late seventies to achieve low latency communications over Ethernet. He added micro-code to the machine to handle time-critical portions in send and receive routines. The major gain seemed to occur by busy waiting on a process expecting a response from a remote machine.

The Message-Driven Processor (MDP), designed by William Daley, has an architecture adapted for efficient message communications in an object-oriented system [19]. A message is considered to be in the form of a long instruction for the processor which can be executed immediately. The architecture supports message handling with an on-chip memory with 3 internal access ports and a hardware support for message queues. Although the design is directed to an object-oriented system, it could also handle a general-purpose transport protocol.

6.4 Host-Network Device Interactions

David Clark from M.I.T. has proposed a host-network interface architecture [15] that resembles in many ways to the host-adapter interface of the NAB. In both works, developed concurrently and independently, the basic premise is the same. Clark also

argues for reducing the load on the host by transferring checksumming, packetization and buffer management functions to an intelligent device. Their proposal does not discuss device architectures necessary for better performance and thus in itself is not sufficient to guarantee high performance. Two other major differences exist as compared to our host-network interface. They envisage the model of interaction to be a virtual circuit; whereas, ours is a simple request-response model that would provide low latency for single packet exchange. The other difference is the proposed global naming for messages transferred across the interface. It would be, in my opinion, difficult to manage global name space of messages leading to a high latency for a single exchange of packets. For this reason we have not named messages explicitly. One of the innovative aspects of their model was how to handle buffer management functions. Their scheme is quite general and would work well with real-time traffic, such as the packetized video. Our buffer management scheme is easier to implement and is as flexible. The NAB architecture provides additional features such as secure communications and traffic overload-protection not provided in their proposal.

6.5 Optimized Transport Protocols

Universal Receiver Protocol [27], developed at Bell Laboratories for the Datakit network, is one of the early attempts at developing a lightweight transport protocol. They noticed that the processing bottleneck is at the receiving end, and thus concentrated on simplifying the URP receiver's state machine. It uses simple packet types and address fields. The packet size is kept fairly small in order to reduce the store-and-forward component of the delay at intermediate switching nodes. Although useful for the environment it was developed, namely centralized-hub type networks, URP seems not flexible enough to operate over subnets with widely different characteristics.

Another example of an light-weight protocol is XTP, proposed by Greg Chesson [14]. The focus here was on simplifying mechanisms such that the entire protocol could be implemented on a VLSI chip. Although the design is somewhat fluid at the moment, the available details suggest that it has focused on simplifying addressing and error-control mechanisms. Although the protocol has more functionality than Sandy Fraser's URP protocol, on which it is based, it still lacks standard transport level features such as support for encryption, multicasting, and process migration. Its error control scheme is a variation on the sliding window control scheme of TCP. It allows only one run of missing packets in the sequence of the received packets, before it would start discarding packets and ask for retransmission beginning with the missing pieces. Because of this variation, the performance is reduced in subnetworks that may reorder packets or split virtual circuit traffic on alternative routes to balance the load on gateways.

The NETBLT is a transport protocol proposed by D. Clark et al. [16] to transfer bulk data. Main feature that was expected to improve throughput was the grouping of packets sent. In NETBLT, the sender sends a large block of data as a group of packets without getting an acknowledgment from the receiver. In order to not overrun the receiver, the sending of a packet group is controlled by introducing gaps in transmitted packets. The rate is negotiated at the opening of a connection and adjusted periodically through feedback from the receiver. NETBLT implemented properly could provide high throughput, but it provides little support for transferring small amounts of data with low latency.

6.6 Current Products

Among the Ethernet devices and other network interfaces commercially available, one finds three basic models. We refer to these as the *memory model*, the *DMA model* and the *processor model*. In the memory model, a host processor reads packets from, and writes packets, to the network adapter as though it is simply a portion of memory, except for performing some operation to restart network packet transmission and reception after every packet. In the DMA model, the board accepts descriptors of packet buffers (possibly as scattered fragments of memory) and performs the packet transfer. In the processor model, the board copies data to and from the host memory plus provides some degree of processing capacity on-board. The VMP interface follows the memory model for small amounts of data and the processor model for large amounts of data.

Several commercial products use the memory model. The 3C400 [1] is an early example, providing only one transmit buffer and two receive buffers.

A more recent and competitive design, the SUN Microsystem's Ethernet interface [59], provides 128 kilobytes of on-board memory plus a network interface chip providing on-board scatter-gather DMA between the network and the on-board memory. One advantage cited for this design is that it allows direct DMA from a disk (for example) to the network interface, providing one bus transfer for data. However, this design assumes no software checksum and no encryption. Inclusion of these features would incur additional memory references. Note also, either all file system buffers reside on the network interface board or else data must be transferred to main memory in any case. With the feasibility of dedicating multi-megabytes of memory to disk cache on a shared file server, the benefits of direct transfers to the network interface seem limited.

Several commercial products use the DMA model, including the DEC board [21]. This design forces a host processor to fully prepare a network packet for transmission in host memory, incurring at least 2 extra memory references and typically 4 if encryption is used.[1] Besides the copying cost, these interfaces interrupt on every packet transmission and reception, placing a significant overhead on the host processor(s). They also do not provide any "firewall" protection between the host and the network. In general, DMA network interfaces do not appear to have any advantages over the memory model.

A number of network interfaces in the processor model are available as commercial products. Examples include the Excelan 1020 [23] and the CMC ENP-100 [18]. In our experience, these interfaces are slower than commercial interfaces in the other models because of inadequate memory bandwidth on the board to serve the network transfers, hosts transfers and on-board processor accesses simultaneously. To be fair, these products are aimed at providing network access for existing operating systems and architectures with minimal modification to the operating system. For instance, the CMC ENP board comes with a TCP implementation that makes it appear similar to a terminal multiplexor.

[1] The author is not aware of any software implementation that performs the copy and checksum simultaneously. In the absence of this optimization, an additional memory reference is required.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we addressed the question about how much transport protocol processing should an adapter do. We justified, qualitatively and quantitatively, the following as the best choice in this regard.

1. **Checksumming and Encryption:** We propose that the adapter perform end-to-end checksum and encryption/decryption functions. If the adapter checksums and encrypts data, then the data movement over the host bus and memory is reduced to one per data word. Results of measurements discussed in Section 5.3 show that 16 Kbytes of data transfer had bus utilization of 70% as compared to that when host performed checksumming and encryption functions. Using the bus efficiently for communication traffic has many benefits. Since data is copied only once, the processing time per packet is reduced. Since a host processor has to wait less often to access bus and memory, the system's overall performance increases. Additionally, one can accommodate a larger number of processors in a bus-based multiprocessor system.

2. **Packet Header Processing:** We propose that the adapter perform the header processing for *intermediate*[1] packets. If the adapter does packet processing, it will generate fewer host interrupts. and it can overlap header processing with the transfer of data over the host bus. Results of measurements, discussed in Section 5.5.3, show that such an adapter has 2.5 ms response time for 16 Kbyte write operation, which is three times faster than if the adapter does not do header processing but does checksumming and encryption. Furthermore, with onboard intelligence (and the buffer reservation model), one does not need physical or virtual copy of data pages between kernel and users' address spaces (Section 3.2.1).

[1] We define an *intermediate* packet as one that does not, explicitly or implicitly, establish or close a transport-level connection.

Current intelligent adapters are slow. This observation motivated us to investigate techniques for improving the performance of an intelligent adapter. We identified three general techniques. The first and the most effective of them is the pipelined processing of packets. The other two techniques are the predictive processing of headers and the optimization for control packets.

1. **Pipelined Processing:** Pipelined processing of a packet reduces memory accessed to the adapter memory, a critical resource at the intelligent adapter. The references are reduced because checksumming and encryption of data is performed while a packet is transmitted or received.

2. **Predictive Processing:** Predicting headers of packets to be received reduces packet processing times. One compares incoming packet header to predicted one(s) instead of doing the costly conditional processing in order to locate connection records and to validate packets. We found that only few predicted entries need to be stored to achieve high success rates for predictions.

3. **Optimization for Control Packets:** Low latency is important for packets carrying control information. Such packets are identified and expedited by the adapter using the programmed I/O model directly to the user.

As a part of our thesis research, we proposed an integrated solution for high performance communication based on the principles identified above. This solution incorporates several novel concepts in the host/adapter interface, the adapter's internal architecture, and the communications protocol architecture. We designed and partially implemented a prototype based on the architecture as a network adapter for the VMP system, a multiprocessor designed at Stanford University. Although the design is applicable to other multiprocessors and network protocols, the conventional independence between network adapter design, computer I/O system design, and transport protocol design seems incompatible with achieving the performance levels of interest here. In the following we summarize some of the important aspects of the design.

Host/Adapter Interface: The host-to-adapter interface treats small and large data transfers differently, handling the small transfers using a programmed I/O interface and the large data transfers with "intelligent" DMA. The programmed I/O interface minimizes delay for small packets. The DMA interface for large transfers minimizes cache and system bus traffic.

The adapter-to-host interface interrupts host on a data segment boundary and not for each packet transferred. For multi-packet transfers, this significantly reduces interrupt processing overhead, which is especially costly in multiprocessor systems, as one may need to trap data and code in the cache to handle interrupts.

The authorization model for reception allows a host to continue functioning even when being bombarded by packets from one or more hosts. This "firewall" function is essential for connecting hosts to a high-speed network.

Network Adapter Internal Architecture: On-board processing of checksumming, encryption, and packetization of data minimizes bus transfers to 1 per unit of transfer, namely a 32-bit word. This also avoids having to transfer data to the processor cache, which may improve the cache performance.

A packet pipeline, executing key functions such as checksumming and encryption, is used to increase throughput, particularly for large data transfers. The pipeline latency for short packet transfers is reduced by using few stages and a small unit for data transfers between the stages.

Contention-less memory accessing is provided by a novel memory architecture based on Video RAMs which reduces buffering latency and increases the packet processing rate. It allows processing of a packet by the on-board processor to proceed in parallel with the transfer of subsequent packets from the host to the buffer memory and from the network to the buffer memory.

Block copier hardware is used to transfer data at full blast between host memory and the adapter memory, thus reducing bus occupancy and buffering latency.

The header processing algorithm utilizes the hints produced by the examination of previously received packets to reduce the packet processing time at a receiver. Our traffic measurements show that the hints succeed for a large fraction of the received packets. We investigated three algorithms for producing hints. Of these three, the one that predicted responses based on outgoing requests produced the best success ratio for a given size of cache containing hints.

Communication Protocol Architecture: VMTP is designed specifically to provide high performance communications. Many features of VMTP are exploited in this design. The packet group concept is exploited by the host/adapter interface to minimize host interrupts. The packet group concept also simplifies the packet formation for packets in the same group. The NAB accepts a packet prototype of the first packet and forms headers of subsequent packets by making simple changes in it. Request-response model is also used to provide hints for reducing packet processing time at a receiver.

The VMTP protocol has two types of packets: fixed size packets containing few words of user-level data and variable size packets containing a large amount of data. This reflects the dual-size distribution observed in network traffic measurements. This feature of VMTP is exploited by the host/adapter interface to efficiently handle both small and large data transfers.

The VMTP checksum, which follows the data field, facilitates checksumming in hardware as the packet is transferred over the network, hid-

ing its cost in a copying operation of transferring the data to the network.

We estimated the performance of the prototype based on timing information from the prototype. The estimated performance is over 45 Mbits/s throughput and 1.2 ms latency, which is at least an order of magnitude higher throughput and 33% less latency than that available from software implementations of VMTP on a Sun 3/75 workstation. Both Sun 3/75 and the prototype uses the same microprocessor, MC68020. Thus, the improvement is seen as the result of improved architecture.

The performance was found to be limited by the processing time of packet headers. The bottleneck was removed in a model of NAB using AMD29000 microprocessor, a 17-MIPS machine. In this set of experiments, the NAB software was run on the chip-level simulator of AMD29000 processor with the parameters that accurately model the NAB adapter's memory. As expected, the measured performance confirmed that the NAB architecture scales up with the increase in processor power. The measured throughput was about two orders of magnitude higher than that of current software implementations of VMTP. Thus, we supported the claim we made earlier that the current general-purpose microprocessors have the power necessary to provide better performance with the integrated changes in system design.

One of the contribution of this work is to clear the myths in designing a transport protocol. As corollaries to this work, we conclude that

1. *High-Performance does not require changing state machines of current transport protocols*
2. *High-Performance does not require that state machines of current transport protocol be implemented in hardware.*
3. *Streamlining of protocols is needed to facilitate the speed-up by using the pipelined processing, the predictive processing, and the optimization for control packets.*

7.2 Future Work

Future work is necessary to address three issues raised by this thesis work.

Fairness among Users of High-Speed Transport If you have a host interface that gives a single user connection the capability to use the full bandwidth of the network link, then it is likely that the user may use it to wreak havoc on the network. Clearly, we need to design transport-level mechanisms that guarantees fairness at transport-level among users sharing the network. Currently we don't even have the capability to indicate user demands of network capacity. Some fairness is ensured currently in local area networks on per node basis. This is implemented at the link-level, e.g., back-off algorithm in Ethernet and TRT and THT timers in a Token Ring. Additional mechanisms for fair allocation of resources at a end-user node need to be developed. Further explorations are necessary also to understand their interactions to the link-level mechanisms, gateway/switch control policies and the flow and error control policies of transport protocol.

Real Issues in Transport Protocol Design As observed earlier, one needs to make few simple changes to current transport protocols to get high performance. This raises a question about what the real issues are in the future design of transport protocol. Much of the work in the past was directed to designing protocols to get high throughput or low latency [16, 27, 14, 6]. Now, it seems, the focus should turn to other issues. Some open questions in this regard are how to provide guarantees on service quality, how to operate over multiple, possibly heterogeneous networks, and how to get around the inherent delay, limited by the speed of light, seen in a wide-area network.

Data Marshalling Transformations of data representations are necessary for communications between heterogeneous computer systems. Examples of such transformations include reversal of byte-order, change of floating-point formats, etc. Such transformations add significantly to the cost of communicating. Even a simple operation like reversal of byte-order of words takes many instructions per word. Other transformations such as changing formats of a real number are more costly. Moreover, with wide-spread networking, we foresee that data transformations

are going to be increasingly frequent. Performing these functions in host software is slow and will offset improvements at transport-level obtained with the NAB architecture. Thus, we see data marshalling as a next group of functions that needs hardware support and if appropriate changes in protocol architecture.

Performing these functions in the packet processing pipeline is an obvious solution. To do so, one needs to design presentation protocols that support hardware implementation. There are two general classes of presentation protocols: explicit and implicit. In an explicit presentation protocol, exemplified by X.409 ISO Abstract Syntax Notation [34], one appends a type field to each data field carried in a packet. Implicit presentation protocols, exemplified by the XDR used within NFS and SunRPC [58], assume that remote application modules have information required to perform these functions and thus avoid carrying type information in a packet. Explicit presentation protocol provides a natural foundation for building data transformation hardware as a part of the packet processing pipeline. But, even an implicit protocol like the XDR protocol provides a way to make it behave like an explicit presentation protocol. Let one assign a number to each data type defined in XDR and add that as a discriminant to a universal type appended to the front of each encoded data type field. This enhanced protocol looks much like an explicit presentation protocol and can be used with the packet processing pipeline. The pipeline need not perform all of the transformations specified in the protocol. One could build efficient hardware for common data transformations and leave the rest to be performed by host or network adapter software. Further explorations and in-depth study are necesary to determine the effectiveness of this approach.

Bibliography

[1] 3-Com Corporation.
3C400 Multibus Ethernet Controller: Reference Manual, 1985.

[2] F. Baskett, J. H. Howard, and J. T. Montague.
Task Communications in DEMOS.
In *Proceedings of the 6th ACM Symposium on Operating System Principles*, pages 23–31, November 16-18 1977.
West Lafayette, Ind.,.

[3] P. Borril and J. Theus.
An Advanced Communication Protocol for the Proposed IEEE 896 Futurebus.
IEEE Micros, 4(4):42–56, August 1984.

[4] J. B. Carter and W. Zwaenepoel.
Implementation of Optimistic Bulk Data Transfer Protocols.
In *SIGMETRICS '89*. ACM, May 1989.
Berkeley, California.

[5] D. R. Cheriton.
The Thoth System: Multi-Process Structuring Portability.
Elssevier/North-Holland, New York, 1982.

[6] D. R. Cheriton.
VMTP: A Transport Protocol for the Next Generation of Communication Systems.
In *Proceedings of SIGCOMM'86*. ACM, Aug 5-7 1986.

[7] D. R. Cheriton.
Exploiting Recursion to Simplify RPC Communication Architectures.
In *SIGCOMM'88*, August 1988.
Stanford, CA.

[8] D. R. Cheriton.
The V Distributed System.
Communications of the ACM, 31(3):314–333, March 1988.

[9] D. R. Cheriton.
Versatile Message Transaction Protocol (VMTP).
RFC 1045, SRI Network Information Center, February 1988.

[10] D. R. Cheriton, G. Slavenberg, and P. Boyle.
Software-Controlled Caches in the VMP Multiprocessor.
Technical Report STAN-CS-86-1105, Dept. of Comp. Sci., Stanford University, 1986.

[11] D. R. Cheriton and C. L. Williamson.
Network Measurement of the VMTP Request-Response Protocol in the V Distributed System.
In *SIGMETRICS '87*. ACM, May 1987.
Banff, Alberta, Canada.

[12] D. R. Cheriton and W. Zwaenepoel.
The Distributed V Kernel and its Performance for Diskless Workstations.
In *Proceedings of 9th Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[13] G. Chesson.
Protocol Engine Project.
Seminar at Stanford University, Stanford, November 1987.

[14] G. Chesson.
XTP/PE Overview.
Seminar at Stanford University, Stanford, March 1988.

[15] D. D. Clark.
Host-Network Interface Architecture.
RFC (draft) 310, M.I.T. Laboratory for Computer Science, 1987.

[16] D. D. Clark, M. Lambert, and L. Zhang.
NETBLT: A Bulk Data Transfer Protocol.
Technical Report RFC 969, Defense Advanced Research Projects Agency, 1985.

[17] D. D. Clark, J. Romkey, and H. Salwen.
An Analysis of TCP Processing Overhead.
In *Proceedings of the 13th Conference on Local Computer Networks*, pages 284–291, October

10-12 1989.
Minneapolis.

[18] Communications Machinery Corp.
Ethernet Interface: User's Guide, 1985.

[19] W. Daley.
Architecture of a Message Driven Processor.
In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987.
Pittsburg, PA.

[20] P. Danzig.
A fine-grained timer hardware for measurements on sun 3 workstations.
University of California, Berkeley.

[21] Digital Equipment Corp.
DEUNA: User's Guide, 1983.

[22] V. Milutinovic et al.
A GaAs-Based Microprocessor Architecture for Real-Time Applications.
IEEE Transactions on Computers, C-36(6), June 1987.

[23] Excelan Corp., Mountain View, Calif.
Excelan 1020: Network Interface - User's Guide, 1985.

[24] *Draft Proposed American National Standard, FDDI Token Ring Media Access Control - ANSC X3T9.5*, 1984.

[25] D. C. Feldmeieir.
Estimated Performance of a Gateway Routing Table Cache.
Technical report, MIT/LCS/TM-352, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1988.

[26] W. Fischer.
IEEE p1014 - A Standard for the High-Performance VME Bus.
IEEE Micros, 5(1), February 1985.

[27] A. G. Fraser, W. Marshal, and G. Riddle.
Proposal for the Universal Receiver Protocol for the Datakit(r) VCS.
TM (draft), AT&T Bell Laboratories, March 1982.

[28] Z. Haas and D. R. Cheriton.
Blazenet: A Photonic Implementable Wide-Area Network.
Technical Report TR-?, Stanford University, 1987.

[29] J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill.
MIPS: A VLSI Processor Architecture.
In *Proceedings CMU Conference on VLSI Systems and Computations*, October 1981.

[30] T. Holman and L. Snyder.
A Transparent Coprocessor for Interprocessor Communiction in an MIMD Computer.
Department of Computer Science TR-87-03-07, University of Washington, Seattle, WA, July 1987.

[31] M. Horowitz and et al.
A 32b Microprocessor with 2K-Byte On-chip Instruction Cache.
In *IEEE International Solid-State Circuits Conference*, February 1987.

[32] H. Ichikawa, H. Yamada, T. Akaike, S. Kanno, and M. Aoki.
Protocol Control VLSI for Broadband Packet Communications.
In *IEEE Global Telecommunications Conference - 88*, pages 1494–98, November 28 1988.
Hollywood, Florida, U.S.A.

[33] Intel.
Multibus II Bus Architecture Specification Handbook, Publication No. 146077B.
Intel Corporation, Literature Department, 3065 Bowers Ave., Santa Clara, CA 95051, 1984.

[34] ISO.
X.409: ISO Abstract Syntax Notation.
Technical report, ISO, ?

[35] Van Jacobson.
Congestion Avoidance and Control.
In *SIGCOMM '88*, pages 314–329. ACM, August 1988.
Stanford University, Stanford, California.

[36] Van Jacobson.
maximum Ethernet throughput.
Letter to the TCP/IP mailing list, March 9 1988.

[37] Van Jacobson.
some interim notes on the bsd network speedups.
Letter to the TCP/IP mailing list, July 19 1988.

[38] R. Jain and S. Routhier.
Packet Trains: Measurements and a New Model for Computer Network Traffic.
IEEE Journal on Selected Areas in Communications, SAC-4(6):986–995, September 1986.

[39] H. Kanakia and D. R. Cheriton.
The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors.
In *SIGCOMM '88*, pages 175–187. ACM, August 1988.
Stanford University, Stanford, California.

[40] H. Kanakia and D. R. Cheriton.
Universal Network Device Interface Protocol - (UNDIP).
In *Proceedings of the 13th Conference on Local Computer Networks*, pages 301–311, October 10-12 1989.
Minneapolis.

[41] H. Kanakia and F. Tobagi.
Performance Measurements of a Data Link Protocol.
In *International Conference on Communications*. IEEE, June 22-25 1986.

[42] A. S. Krishnakumar, B. Krishnamurthy, and K. Subnani.
Translation of Formal Protocol Specifications to VLSI Design.
Elsevier Science Publishers B.V.(North-Holland), 1987.

[43] K. A. Lantz, W. I. Nowicki, and M. M. Theimer.
An Empirical Study of Distributed Application Performance.
IEEE Transactions on Software Engineering, SE-11(10):1162–1174, October 1985.

[44] J. Limb.
Fasnet: A Unidirectional High Speed Network.
BSTJ, 1985.

[45] M. J. Lorence and M. Satyanarayanan.
IPwatch: A Tool for Monitoring Network Locality.
Technical report, Department of Computer Science, Carnegie-Mellon University, December 1988.

[46] Quanta Microtique.
Preliminary Report on the QM10 advanced communications controller.
Washington, D.C., 1983.

[47] Jean-Daniel Nicoud.
Video RAMs: Structure and Applications.
IEEE Micro, pages 8–27, February 1988.

[48] E. Nordamrk and D. R. Cheriton.
Experiences from VMTP: How to Achieve Low Response Time.
Presented at IFIP/WG6.1-WG6.4 Workshop on Protocols for High-Speed Networks. Zurich., May 9-12 1989.

[49] J. Ousterhout, H. Da Costa, D. Harrison, J. kunze, M. Kupfer, and J. Thompson.
A Trace-driven Analysis of the Unix 4.2 BSD File System.
In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[50] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel.
Locus: A Network Transparent, High reliability Distributed System.
In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 169–177. ACM, December 1981.

[51] U. Ramachandran, M. Solomon, and M. Vernon.
Hardware Support for Interprocess Communications.
In *Proceedings of 14th International Symposium on Computer Architecture*, pages 178–188, June 1987.

[52] R. Rashid and G. G. Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[53] D. P. Reed, J. H. Saltzer, and D. D. Clark.
End-to-End Arguments in System Design.
ACM Transactions on Computer Systems, 2(4):277–288, November 1984.

[54] K. Sabnani and A. Lapone.
PAV - Protocol Analyzer and Verifier.
Elsevier Science Publishers B.V.(North-Holland), 1986.

[55] J. F. Shoch and J. A. Hupp.
Measured Performance of an Ethernet Local Network.
Communications of the ACM, 23(12):711–721, December 1980.

[56] IEEE Computer Society.
Standards for Local Area Networks: Logical Link Control.
Technical report, ISO/ANSI, 1985.

ANSI/IEEE Standard 802.2-1985 (ISO/DIS 8802/2).

[57] A. Z. Spector.
Multiprocessing Architectures for Local Computer Networks.
Technical Report STAN-CS-81-874, Department of Computer Science, Stanford University, 1981.

[58] Inc. Sun Microsystems.
XDR: External Data Representation Standard.
RFC 1014, Defense Advanced Research Projects Agency, June 1987.

[59] SUN Microsystems, Corp.
SUN-3 Architecture: A Sun Technical Report, 1985.

[60] Garret Swart.
Personal communication.
Digital Equipment Corporation, Systems Research Center.

[61] C. Thacker.
Personal communciation.
Digital Equipment Corporation, Systems Research Center.

[62] C. L. Williamson.
Dynamic Transport-Level Connection Management in a Distributed System.
In *14th Conference on Local Computer Networks*, October 1989.
Minneapolis.